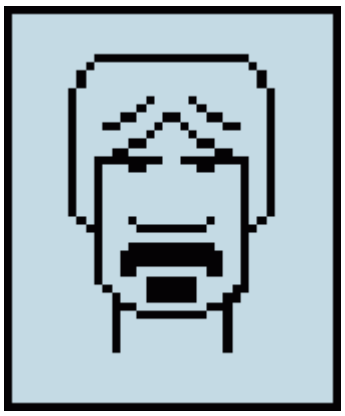
Java 反射机制文章集合

关键字：Java 反射 Class



由“彭益存”从网络整理

Email: yicun05@software.nju.edu.cn

pengyicun05@163.com

2009/03/25

目录

Java 反射机制

反射的概念 ：

反射的概念是由 Smith 在1982年首次提出的，主要是指程序可以访问、检测和修改它本身状态或行为的一种能力。这一 概念的提出很快引发了计算机科学领域关于应用反射性的研究。它首先被程序语言的设计领域所采用,并在 Lisp 和面向对象方面取得了成绩。其中 LEAD/LEAD++ 、OpenC++ 、MetaXa 和 OpenJava 等就是基于反射机制的语言。最近，反射机制也被应用到了视窗系统、操作系统和文件系统中。

反射本身并不是一个新概念，它可能会使我们联想到光学中的反射概念，尽管计算机科学赋予了反射概念新的含义，但是，从现象上来说，它们确实有某些相 通之处，这些有助于我们的理解。在计算机科学领域，反射是指一类应用，它们能够自描述和自控制。也就是说，这类应用通过采用某种机制来实现对自己行为的描 述（ self-representation ）和监测（examination），并能根据自身行为的状态和结果，调整或修改应用所描述行为的状态和相关 的语义。可以看出，同一般的反射概念相比，计算机科学领域的反射不单单指反射本身，还包括对反射结果所采取的措施。所有采用反射机制的系统（即反射系统） 都希望使系统的实现更开放。可以说，实现了反射机制的系统都具有开放性，但具有开放性的系统并不一定采用了反射机制，开放性是反射系统的必要条件。一般来 说，反射系统除了满足开放性条件外还必须满足原因连接（Causally-connected）。所谓原因连接是指对反射系统自描述的改变能够立即反映到 系统底层的实际状态和行为上的情况，反之亦然。开放性和原因连接是反射系统的两大基本要素。13700863760

Java 中，反射是一种强大的工具。它使您能够创建灵活的代码，这些代码可以在运行时装配，无需在组件之间进行源代表链接。反射允许我们在编写与执 行时，使我们的程序代码能够接入装载到 JVM 中的类的内部信息，而不是源代码中选定的类协作的代码。这使反射成为构建灵活的应用的主要工具。但需注意的 是：如果使用不当，反射的成本很高。

Java 中的类反射：

Reflection 是 Java 程序开发语言的特征之一，它允许运行中的 Java 程序对自身进行检查，或者说“自审”，并能直接操作程序的内部属性。Java 的这一能力在实际应用中也许用得不是很多，但是在其它的程序设计语言中根本就不存在这一特性。例如，Pascal、C 或者 C++ 中就没有办法在程序中获得函数定义相关的信息。

1．检测类：

1.1 reflection 的工作机制

考虑下面这个简单的例子，让我们看看 reflection 是如何工作的。

```
import java.lang.reflect.*;
public class DumpMethods {
      public static void main(String args[]) {
            try {
                  Class c = Class.forName(args[0]);
                  Method m[] = c.getDeclaredMethods();
                  for (int i = 0; i < m.length; i++)
                        System.out.println(m[i].toString());
            } catch (Throwable e) {
                  System.err.println(e);
            }
      }
}
```

按如下语句执行：

```
java DumpMethods java.util.Stack
```

它的结果输出为：

```
public java.lang.Object java.util.Stack.push(java.lang.Object)

public synchronized java.lang.Object java.util.Stack.pop()

public synchronized java.lang.Object java.util.Stack.peek()

public boolean java.util.Stack.empty()

public synchronized int java.util.Stack.search(java.lang.Object)
```

这样就列出了 java.util.Stack 类的各方法名以及它们的限制符和返回类型。

这个程序使用 Class.forName 载入指定的类，然后调用 getDeclaredMethods 来获取这个类中定义了的方法列表。java.lang.reflect.Methods 是用来描述某个类中单个方法的一个类。

1.2 Java 类反射中的主要方法

对于以下三类组件中的任何一类来说 -- 构造函数、字段和方法 -- java.lang.Class 提供四种独立的反射调用，以不同的方式来获得信息。调用都遵循一种标准格式。以下是用于查找构造函数的一组反射调用：

l Constructor getConstructor(Class[] params) -- 获得使用特殊的参数类型的公共构造函数，

l Constructor[] getConstructors() -- 获得类的所有公共构造函数

l Constructor getDeclaredConstructor(Class[] params) -- 获得使用特定参数类型的构造函数(与接入级别无关)

l Constructor[] getDeclaredConstructors() -- 获得类的所有构造函数 (与接入级别无关)

获得字段信息的 Class 反射调用不同于那些用于接入构造函数的调用，在参数类型数组中使用了字段名：

l Field getField(String name) -- 获得命名的公共字段

l Field[] getFields() -- 获得类的所有公共字段

l Field getDeclaredField(String name) -- 获得类声明的命名的字段

l Field[] getDeclaredFields() -- 获得类声明的所有字段

用于获得方法信息函数：

l Method getMethod(String name, Class[] params) -- 使用特定的参数类型，获得命名的公共方法

l Method[] getMethods() -- 获得类的所有公共方法

l Method getDeclaredMethod(String name, Class[] params) -- 使用特写的参数类型，获得类声明的命名的方法

l Method[] getDeclaredMethods() -- 获得类声明的所有方法

1.3 开始使用 Reflection：

用于 reflection 的类，如 Method，可以在 java.lang.relfect 包中找到。使用这些类的时候必须要遵循三个步骤：第一步是获得你想操作的类的 java.lang.Class 对象。在运行中的 Java 程序中，用 java.lang.Class 类来描述类和接口等。

下面就是获得一个 Class 对象的方法之一：

```
Class c = Class.forName("java.lang.String");
```

这条语句得到一个 String 类的类对象。还有另一种方法，如下面的语句：

```
Class c = int.class;
```

或者

```
Class c = Integer.TYPE;
```

它们可获得基本类型的类信息。其中后一种方法中访问的是基本类型的封装类 (如 Integer) 中预先定义好的 TYPE 字段。

第二步是调用诸如 getDeclaredMethods 的方法，以取得该类中定义的所有方法的列表。

一旦取得这个信息，就可以进行第三步了——使用 reflection API 来操作这些信息，如下面这段代码：

```
Class c = Class.forName("java.lang.String");

Method m[] = c.getDeclaredMethods();

System.out.println(m[0].toString());
```

它将以文本方式打印出 String 中定义的第一个方法的原型。

2．处理对象：

如果要作一个开发工具像 debugger 之类的，你必须能发现 filed values,以下是三个步骤:

a.创建一个 Class 对象
b.通过 getField 创建一个 Field 对象
c.调用 Field.getXXX(Object)方法(XXX 是 Int,Float 等，如果是对象就省略；Object 是指实例).

例如：

```
import java.lang.reflect.*;
```

```
import java.awt.*;

class SampleGet {

    public static void main(String[] args) {
        Rectangle r = new Rectangle(100, 325);
        printHeight(r);

    }

    static void printHeight(Rectangle r) {
        Field heightField;
        Integer heightValue;
        Class c = r.getClass();
        try {
            heightField = c.getField("height");
            heightValue = (Integer) heightField.get(r);
            System.out.println("Height: " + heightValue.toString());
        } catch (NoSuchFieldException e) {
                System.out.println(e);
        } catch (SecurityException e) {
                System.out.println(e);
        } catch (IllegalAccessException e) {
                System.out.println(e);
        }
    }
}
```

安全性和反射：

在处理反射时安全性是一个较复杂的问题。反射经常由框架型代码使用，由于这一点，我们可能希望框架能够全面接入代码，无需考虑常规的接入限制。但是，在其它情况下，不受控制的接入会带来严重的安全性风险，例如当代码在不值得信任的代码共享的环境中运行时。

由于这些互相矛盾的需求，Java 编程语言定义一种多级别方法来处理反射的安全性。基本模式是对反射实施与应用于源代码接入相同的限制：

n 从任意位置到类公共组件的接入

n 类自身外部无任何到私有组件的接入

n 受保护和打包（缺省接入）组件的有限接入

不过至少有些时候，围绕这些限制还有一种简单的方法。我们可以在我们所写的类中，扩展一个普通的基本类 java.lang.reflect.AccessibleObject 类。这个类定义了一种 setAccessible 方法，使我们能够启动或关闭对这些类中其中一个类的实例的接入检测。唯一的问题在于如果使用了安全性管理 器，它将检测正在关闭接入检测的代码是否许可了这样做。如果未许可，安全性管理器抛出一个例外。

下面是一段程序，在 TwoString 类的一个实例上使用反射来显示安全性正在运行：

```
public class ReflectSecurity {

    public static void main(String[] args) {

        try {

            TwoString ts = new TwoString("a", "b");

            Field field = clas.getDeclaredField("m_s1");

//              field.setAccessible(true);

            System.out.println("Retrieved value is " +

                field.get(inst));

        } catch (Exception ex) {

            ex.printStackTrace(System.out);

        }

    }

}
```

如果我们编译这一程序时，不使用任何特定参数直接从命令行运行，它将在 field .get(inst)调用中抛出一个 IllegalAccessException 异常。如果我们不注释 field.setAccessible (true)代码行，那么重新编译并重新运行该代码，它将编译成功。最后，如果我们在命令行添加了 JVM 参数- Djava.security.manager 以实现安全性管理器，它仍然将不能通过编译，除非我们定义了 ReflectSecurity 类的许可权限。

反射性能：

反射是一种强大的工具，但也存在一些不足。一个主要的缺点是对性能有影响。使用反射基本上是一种解释操作，我们可以告诉 JVM，我们希望做什么并且它满足我们的要求。这类

操作总是慢于只直接执行相同的操作。

下面的程序是字段接入性能测试的一个例子，包括基本的测试方法。每种方法测试字段接入的一种形式 -- accessSame 与同一对象的成员字段协作，accessOther 使用可直接接入的另一对象的字段， accessReflection 使用可通过反射接入的另一对象的字段。在每种情况下，方法执行相同的计算 -- 循环中简单的加/乘顺序。

程序如下：

```
public int accessSame(int loops) {

    m_value = 0;

    for (int index = 0; index < loops; index++) {

        m_value = (m_value + ADDITIVE_VALUE) *

            MULTIPLIER_VALUE;

    }

    return m_value;

}


public int accessReference(int loops) {

    TimingClass timing = new TimingClass();

    for (int index = 0; index < loops; index++) {

        timing.m_value = (timing.m_value + ADDITIVE_VALUE) *

            MULTIPLIER_VALUE;

    }

    return timing.m_value;

}


public int accessReflection(int loops) throws Exception {
```

```
        TimingClass timing = new TimingClass();
        try {
            Field field = TimingClass.class.
                getDeclaredField("m_value");
            for (int index = 0; index < loops; index++) {
                int value = (field.getInt(timing) +
                    ADDITIVE_VALUE) * MULTIPLIER_VALUE;
                field.setInt(timing, value);
            }
            return timing.m_value;
        } catch (Exception ex) {
            System.out.println("Error using reflection");
            throw ex;
        }
}
```

在上面的例子中，测试程序重复调用每种方法，使用一个大循环数，从而平均多次调用的时间衡量结果。平均值中不包括每种方法第一次调用的时间，因此初始化时间不是结果中的一个因素。下面的图清楚的向我们展示了每种方法字段接入的时间：

图 1：字段接入时间 ：

我们可以看出：在前两副图中(Sun JVM)，使用反射的执行时间超过使用直接接入的1000倍以上。通过比较，IBM JVM 可能稍好一些，但反射方法仍旧需要比其它方法长700倍以上的时间。任何 JVM 上其它两种方法之间时间方面无任何显著差异，但 IBM JVM 几乎比 Sun JVM 快一倍。最有可能的是这种差异反映了 Sun Hot Spot JVM 的专业优化，它在简单基准方面表现得很糟糕。反射性能是 Sun 开发1.4 JVM 时关注的一个方面，它在反射方法调用结果中显示。在这类操作的性能方面，Sun 1.4.1 JVM 显示了比1.3.1版本很大的改进。

如果为为创建使用反射的对象编写了类似的计时测试程序，我们会发现这种情况下的差异不象字段和方法调用情况下那么显著。使用 newInstance ()调用创建一个简单的 java.lang.Object

实例耗用的时间大约是在 Sun 1.3.1 JVM 上使用 new Object()的12倍，是在 IBM 1.4.0 JVM 的四倍，只是 Sun 1.4.1 JVM 上的两部。使用 Array.newInstance(type, size)创建一个数组耗用的时间是任何测试的 JVM 上使用 new type[size]的两倍，随着数组大小的增加，差异逐步缩小。

结束语：

Java 语言反射提供一种动态链接程序组件的多功能方法。它允许程序创建和控制任何类的对象(根据安全性限制)，无需提前硬编码目 标类。这些特性使得反射特别适用于创建以非常普通的方式与对象协作的库。例如，反射经常在持续存储对象为数据库、XML 或其它外部格式的框架中使用。 Java reflection 非常有用，它使类和数据结构能按名称动态检索相关信息，并允许在运行着的程序中操作这些信息。Java 的这一特性非常强大，并且是其它一些常用语言，如 C、C++、Fortran 或者 Pascal 等都不具备的。

但反射有两个缺点。第一个是性能问题。用于字段和方法接入时反射要远慢于直接代码。性能问题的程度取决于程序中是如何使用反射的。如果它作为程序运 行中相对很少涉及的部分，缓慢的性能将不会是一个问题。即使测试中最坏情况下的计时图显示的反射操作只耗用几微秒。仅反射在性能关键的应用的核心逻辑中使 用时性能问题才变得至关重要。

许多应用中更严重的一个缺点是使用反射会模糊程序内部实际要发生的事情。程序人员希望在源代码中看到程序的逻辑，反射等绕过了源代码的技术会带来维 护问题。反射代码比相应的直接代码更复杂，正如性能比较的代码实例中看到的一样。解决这些问题的最佳方案是保守地使用反射——仅在它可以真正增加灵活性的 地方——记录其在目标类中的使用。

利用反射实现类的动态加载

Bromon 原创 请尊重版权

最近在成都写一个移动增值项目，俺负责后台 server 端。功能很简单，手机用户通过 GPRS 打开 Socket 与服务器连接，我则根据用户传过来的 数据做出响应。做过类似项目的兄弟一定都知道，首先需要定义一个类似于 MSNP 的通讯协议，不过今天的话题是如何把这个系统设计得具有高度的扩展性。由于 这个项目本身没有进行过较为完善的客户沟通和需求分析，所以以后肯定会有很多功能上的扩展，通讯协议肯定会越来越庞大，而我作为一个不那么勤快的人，当然 不想以后再去修改写好的程序，所以这个项目是实践面向对象设计的好机会。

首先定义一个接口来隔离类：

```
package org.bromon.reflect;

public interface Operator
```

```
{

public java.util.List act(java.util.List params)

}
```

根据设计模式的原理，我们可以为不同的功能编写不同的类，每个类都继承 Operator 接口，客户端只需要针对 Operator 接口编程就可以避免很多麻烦。比如这个类：

```
package    org.bromon.reflect.*;

import    java.util.*;

public class Success implements Operator

{

public java.util.List act(java.util.List params)

{

List result=new ArrayList();

result.add(new String("操作成功"));

return result;

}

}
```

我们还可以写其他很多类，但是有个问题，接口是无法实例化的，我们必须手动控制具体实例化哪个类，这很不爽，如果能够向应用程序传递一个参数，让自己去选择实例化一个类，执行它的 act 方法，那我们的工作就轻松多了。

很幸运，我使用的是 Java，只有 Java 才提供这样的反射机制，或者说内省机制，可以实现我们的无理要求。编写一个配置文件 emp.properties:

```
#成功响应

1000=Success

#向客户发送普通文本消息

2000=Load
```

#客户向服务器发送普通文本消息

3000=Store

文件中的键名是客户将发给我的消息头，客户发送1000给我，那么我就执行 Success 类的 act 方法，类似的如果发送2000给我，那就执行 Load 类的 act 方法，这样一来系统就完全符合开闭原则了，如果要添加新的功能，完全不需要修改已有代码，只需要在配置文件中添加对应规则，然后编写新 的类，实现 act 方法就 ok，即使我弃这个项目而去，它将来也可以很好的扩展。这样的系统具备了非常良好的扩展性和可插入性。

下面这个例子体现了动态加载的功能，程序在执行过程中才知道应该实例化哪个类：

```
package org.bromon.reflect.*;

import java.lang.reflect.*;

import java.io.*;

import java.util.*;

public class TestReflect

{

//加载配置文件,查询消息头对应的类名

private String loadProtocal(String header)

{

String result=null;

try

{

Properties prop=new Properties();

FileInputStream fis=new FileInputStream("emp.properties");

prop.load(fis);

result=prop.getProperty(header);

fis.close();

}catch(Exception e)
```

```
{
System.out.println(e);
}
return result;
}
//针对消息作出响应,利用反射导入对应的类
public String response(String header,String content)
{
String result=null;
String s=null;
try
{
/*
*  导入属性文件 emp.properties,查询 header 所对应的类的名字
*  通过反射机制动态加载匹配的类,所有的类都被 Operator 接口隔离
*  可以通过修改属性文件、添加新的类(继承 MsgOperator 接口)来扩展协议
*/
s="org.bromon.reflect."+this.loadProtocal(header);
//加载类
Class c=Class.forName(s);
//创建类的事例
Operator mo=(Operator)c.newInstance();
//构造参数列表
Class params[]=new Class[1];
```

```
params[0]=Class.forName("java.util.List");
//查询 act 方法
Method m=c.getMethod("act",params);
Object args[]=new Object[1];
List    array=new ArrayList<String>();
array.add(content);
args[0]=array;
//调用方法并且获得返回
Object returnObject=m.invoke(mo,args);
System.out.println(returnObject);
}catch(Exception e)
{
System.out.println("Handler-response:"+e);
}
return result;
}
public static  void main(String args[])
{
TestReflect tr=new TestReflect();
tr.response(args[0],"消息内容");
}
}
```

测试一下：java TestReflect 1000

这个程序是针对 Operator 编程的，所以无需做任何修改，直接提供 Load 和 Store 类，就可

以支持2000、3000做参数的调用。

有了这样的内省机制，可以把接口的作用发挥到极至，设计模式也更能体现出威力，而不仅仅供我们饭后闲聊。

谈 java 反射机制

本文来源于网络，如果作者不允许我转，请与我联系，我删除就是:)

Person p=new Person();

这是什么?当然是实例化一个对象了.可是这种实例化对象的方法存在一个问题,那就是必须要知道类名才可以实例化它的对象,这样我们在应用方面就会受到限制. 那么有没有这样一种方式,让我们不知道这个类的类名就可以实例化它的对象呢?Thank Goodness!幸亏我们用的是 java, java 就提供了这样的机制.

1).java 程序在运行时可以获得任何一个类的字节码信息,包括类的修饰符(public,static 等),基类(超类,父类),实现的接口,字段和方法等信息.

2).java 程序在运行时可以根据字节码信息来创建该类的实例对象,改变对象的字段内容和调用对象方法.

这样的机制就叫反射技术.可以想象光学中的反射,就像我们照镜子,镜子中又出现一个自己(比喻可能不太恰当,但是足以表达清楚意思了).反射技术提供了一种通用的动态连接程序组件的方法,不必要把程序所需要的目标类硬编码到源程序中,从而使得我们可以创建灵活的程序.

Java 的反射机制是通过反射 API 来实现的,它允许程序在运行过程中取得任何一个已知名称的类的内部信息.反射 API 位于 java.lang.reflect 包中.主要包括以下几类:

1).Constructor 类:用来描述一个类的构造方法
2).Field 类:用来描述一个类的成员变量
3).Method 类:用来描述一个类的方法.
4).Modifer 类:用来描述类内各元素的修饰符
5).Array:用来对数组进行操作.

Constructor, Field,Method 这三个类都是 JVM(虚拟机)在程序运行时创建的,用来表示加载类中相应的成员 .这三个类都实现了 java.lang.reflect.Member 接口,Member 接口定义了获取类成员或构造方法等信息的方法.要使用这些反射 API,必须先得到要 操作的对象或类的 Class 类的实例.通过调用 Class 类的 newInstance 方法(只能调用类的默认构造方法)可以创建类的实例.这样有局限性, 我们可以先冲类的 Class 实例获取类需要的构造方法,然后在利用反射来创建类的一个实例.

一.获取类的构造方法的 Constructor 对象(数组)

● Constructor[] getDeclaredConstructors();返回已加载类声明的所有的构造方法的 Constructor 对象数组.

● Constructor getDeclaredConstructor(Class[] paramTypes);返回已加载类声明的指定构造方法的 Constructor 对象,paramTypes 指定了参数类型.

● Constructor[] getConstructors();返回已加载类声明的所有的 public 类型的构造方法的 Constructor 对象数组.

● Constructor getConstructor(Class[] paramTypes);返回已加载类声明的指定的 public 类型的构造方法的 Constructor 对象,paramTypes 指定了参数类型.

如果某个类中没有定义构造方法,第一个和第三个方法返回的数组中只有一个元素,就是缺省的构造方法;如果某个类中只定义了有参数的构造函数,而没有定义缺省构造函数,第一个和第三个方法返回的数组中不包含缺省的构造方法.

例子:

```
import java.lang.reflect.*;
public class DumpMethods {
    public static void main(String[] args) {
        try{
                if(args.length<1){
                    System.out.println("请输入完整的类名：");
                        return;
                }
        Class strClass=Class.forName(args[0]);
        //检索带有指定参数的构造方法
        Class[] strArgsClass=new Class[]{ byte[].class,String.class};
        Constructor constructor=strClass.getConstructor(strArgsClass);
        System.out.println("Constructor:"+constructor.toString());

        //调用带有参数的构造方法创建实例对象 object
        byte[] bytes="java 就业培训".getBytes();
        Object[] strArgs=new Object[]{bytes,"gb2312"};
        Object object=constructor.newInstance(strArgs);
        System.out.println("Object"+object.toString());
        }catch(Exception e){
            e.printStackTrace();
        }
    }
}
```

运行结果:

二.获取类成员变量的 Field 对象(数组)

●Field[] getDeclaredFields():返回已加载类声明的所有成员变量的Field 对象数组,不包括从父类继承的成员变量.

●Field getDeclaredField(String name): 返回已加载类声明的所有成员变量的 Field 对象,不包括从父类继承的成员变量,参数 name 指定成员变量的名称.

●Field[] getFields():返回已加载类声明的所有 public 型的成员变量的 Field 对象数组,包括从父类继承的成员变量

●Field getField(String name):返回已加载类声明的所有成员变量的 Field 对象,包括从父类继承的成员变量,参数 name 指定成员变量的名称.

例子:

```
import java.lang.reflect.*;
public class ReflectTest {
	private String name;
	private String age;
	public ReflectTest(String name,String age){
		this.name=name;
		this.age=age;
	}

	public static void main(String[] args) {
		// TODO 自动生成方法存根
		try{
			ReflectTest rt=new ReflectTest("zhanghandong","shiba");
			fun(rt);
		}catch(Exception e){
			e.printStackTrace();
		}

	}
	public static void fun(Object obj) throws Exception{
		Field[] fields=obj.getClass().getDeclaredFields();
		System.out.println("替换之前的:");
```

```
        for(Field field:fields){
            System.out.println(field.getName()+"="+field.get(obj));
            if(field.getType().equals(java.lang.String.class)){
                field.setAccessible(true);
                //必须设置为 true 才可以修改成员变量
                String org=(String)field.get(obj);
                  field.set(obj,org.replaceAll("a","b"));
               }
             }
             System.out.println("替换之后的：");
      for(Field field:fields){
                System.out.println(field.getName()+"="+field.get(obj));
       }
       }
}
```

运行结果如下:

三.获取类的方法的 Method 对象(数组)

●Method[] getDeclaredMethods():返回已加载类声明的所有方法的 Method 对象数组,不包括从父类继承的方法.

●Method getDeclaredMethod(String name,Class[] paramTypes):返回已加载类声明的所有方法的 Method 对象,不包括从父类继承的方法,参数 name 指定方法的名称,参数 paramTypes 指定方法的参数类型.

●Method[] getMethods():返回已加载类声明的所有方法的 Method 对象数组,包括从父类继承的方法.

●Method getMethod(String name,Class[] paramTypes):返回已加载类声明的所有方法的 Method 对象,包括从父类继承的方法,参数 name 指定方法的名称,参数 paramTypes 指定方法的参数类型.

四.检索类的其他信息

- int getModifiers():返回已加载类的修饰符的整形标识值.
- Package getPackage():返回已加载类的包名
- Class getSuperclass(): 返回已加载类的父类的 Class 实例.
- Class [] getInterfaces():返回已加载类实现的接口的 Class 对象数组.
- boolean isInterface(): 返回已加载类是否是接口.

反射的功能很强大,但是使用不当可能会缺点大于优点,反射使代码逻辑混乱,会带来维护的问题.

Java 反射机制

反射机制:所谓的反射机制就是 java 语言在运行时拥有一项自观的能力。通过这种能力可以彻底的了解自身的情况为下一步的动作做准备。下面具体介绍一下 java 的反射机制。这里你将颠覆原来对 java 的理解。
Java 的反射机制的实现要借助于 4 个类：class，Constructor，Field，Method；其中 class 代表的时类对 象，Constructor－类的构造器对象，Field－类的属性对象，Method－类的方法对象。通过这四个对象我们可以粗略的看到一个类的各个组 成部分。

Class：程序运行时，java 运行时系统会对所有的对象进行运行时类型的处理。这项信息记录了每个对象所属的类，虚拟机通常使用运行时类型信息选择正 确的方法来执行（摘自：白皮书）。但是这些信息我们怎么得到啊，就要借助于 class 类对象了啊。在 Object 类中定义了 getClass()方法。我 们可以通过这个方法获得指定对象的类对象。然后我们通过分析这个对象就可以得到我们要的信息了。

比如：ArrayList arrayList;

Class clazz = arrayList.getClass();

然后我来处理这个对象 clazz。

当然了 Class 类具有很多的方法，这里重点将和 Constructor，Field，Method 类有关系的方法。

Reflection 是 Java 程序开发语言的特征之一，它允许运行中的 Java 程序对自身进行检查，或者说“自审”，并能直接操作程序的内部属性。Java 的这一能力在实际应用中也许用得不是很多，但是个人认为要想对 java 有个更加深入的了解还是应该掌握的。

1．检测类：

reflection 的工作机制

考虑下面这个简单的例子，让我们看看 reflection 是如何工作的。

```
import java.lang.reflect.*;

public class DumpMethods {
```

```
public static void main(String args[]) {

try {

Class c = Class.forName(args[0]);

Method m[] = c.getDeclaredMethods();

for (int i = 0; i < m.length; i++)

System.out.println(m[i].toString());

} catch (Throwable e) {

System.err.println(e);

}

}

}
```

按如下语句执行：

java DumpMethods java.util.ArrayList

这个程序使用 Class.forName 载入指定的类，然后调用 getDeclaredMethods 来获取这个类中定义了的方法列表。java.lang.reflect.Methods 是用来描述某个类中单个方法的一个类。

Java 类反射中的主要方法

对于以下三类组件中的任何一类来说 -- 构造函数、字段和方法 -- java.lang.Class 提供四种独立的反射调用，以不同的方式来获得信息。调用都遵循一种标准格式。以下是用于查找构造函数的一组反射调用：

Constructor getConstructor(Class[] params) -- 获得使用特殊的参数类型的公共构造函数，

Constructor[] getConstructors() -- 获得类的所有公共构造函数

Constructor getDeclaredConstructor(Class[] params) -- 获得使用特定参数类型的构造函数(与接入级别无关)

Constructor[] getDeclaredConstructors() -- 获得类的所有构造函数(与接入级别无关)

获得字段信息的 Class 反射调用不同于那些用于接入构造函数的调用，在参数类型数组中使用了字段名：

Field getField(String name) -- 获得命名的公共字段

Field[] getFields() -- 获得类的所有公共字段

Field getDeclaredField(String name) -- 获得类声明的命名的字段

Field[] getDeclaredFields() -- 获得类声明的所有字段

用于获得方法信息函数：

Method getMethod(String name, Class[] params) -- 使用特定的参数类型，获得命名的公共方法

Method[] getMethods() -- 获得类的所有公共方法

Method getDeclaredMethod(String name, Class[] params) -- 使用特写的参数类型，获得类声明的命名的方法

Method[] getDeclaredMethods() -- 获得类声明的所有方法

使用 Reflection：

用于 reflection 的类，如 Method，可以在 java.lang.relfect 包中找到。使用这些类的时候必须要遵循三个步骤：第一步是获得你想操作的类的 java.lang.Class 对象。在运行中的 Java 程序中，用 java.lang.Class 类来描述类和接口等。

下面就是获得一个 Class 对象的方法之一：

Class c = Class.forName("java.lang.String");

这条语句得到一个 String 类的类对象。还有另一种方法，如下面的语句：

Class c = int.class;

或者

Class c = Integer.TYPE;

它们可获得基本类型的类信息。其中后一种方法中访问的是基本类型的封装类 (如 Intege) 中预先定义好的 TYPE 字段。

第二步是调用诸如 getDeclaredMethods 的方法，以取得该类中定义的所有方法的列表。

一旦取得这个信息，就可以进行第三步了——使用 reflection API 来操作这些信息，如下面这段代码：

```
Class c = Class.forName("java.lang.String");

Method m[] = c.getDeclaredMethods();

System.out.println(m[0].toString());
```

它将以文本方式打印出 String 中定义的第一个方法的原型。

处理对象：

a.创建一个 Class 对象

b.通过 getField 创建一个 Field 对象

c.调用 Field.getXXX(Object)方法(XXX 是 Int,Float 等，如果是对象就省略；Object 是指实例).

例如：

```
import java.lang.reflect.*;

import java.awt.*;

class SampleGet {

public static void main(String[] args) {

Rectangle r = new Rectangle(100, 325);

printHeight(r);

}

static void printHeight(Rectangle r) {

Field heightField;

Integer heightValue;

Class c = r.getClass();
```

```
try {

heightField = c.getField("height");

heightValue = (Integer) heightField.get(r);

System.out.println("Height: " + heightValue.toString());

} catch (NoSuchFieldException e) {

System.out.println(e);

} catch (SecurityException e) {

System.out.println(e);

} catch (IllegalAccessException e) {

System.out.println(e);

}

}

}
```

安全性和反射：

在处理反射时安全性是一个较复杂的问题。反射经常由框架型代码使用，由于这一点，我们可能希望框架能够全面接入代码，无需考虑常规的接入限制。但是，在其它情况下，不受控制的接入会带来严重的安全性风险，例如当代码在不值得信任的代码共享的环境中运行时。

由于这些互相矛盾的需求，Java 编程语言定义一种多级别方法来处理反射的安全性。基本模式是对反射实施与应用于源代码接入相同的限制：

从任意位置到类公共组件的接入

类自身外部无任何到私有组件的接入

受保护和打包（缺省接入）组件的有限接入

不过至少有些时候，围绕这些限制还有一种简单的方法。我们可以在我们所写的类中，扩展一个普通的基本类 java.lang.reflect.AccessibleObject 类。这个类定义了一种 setAccessible 方法，使我们能够启动或关闭对这些类中其中一个类的实例的接入检测。唯一的问题在于如果使用了安全性管理 器，它将检测正在关闭接入检测的代码是否许可了这样做。如果未许可，安全性管理器抛出一个例外。

下面是一段程序，在 TwoString 类的一个实例上使用反射来显示安全性正在运行：

```
public class ReflectSecurity {

public static void main(String[] args) {

try {

TwoString ts = new TwoString("a", "b");

Field field = clas.getDeclaredField("m_s1");

// field.setAccessible(true);

System.out.println("Retrieved value is " +

field.get(inst));

} catch (Exception ex) {

ex.printStackTrace(System.out);

}

}

}
```

如果我们编译这一程序时，不使用任何特定参数直接从命令行运行，它将在 field .get(inst) 调用中抛出一个 IllegalAccessException 异常。如果我们不注释 field.setAccessible(true)代码行，那么重新编译并重新运行该代码，它将编译成功。最后，如果我们在命令行添加了 JVM 参数 -Djava.security.manager 以实现安全性管理器，它仍然将不能通过编译，除非我们定义了 ReflectSecurity 类的许可权 限。

反射性能：（转录别人的啊）

反射是一种强大的工具，但也存在一些不足。一个主要的缺点是对性能有影响。使用反射基本上是一种解释操作，我们可以告诉 JVM，我们希望做什么并且它满足我们的要求。这类操作总是慢于只直接执行相同的操作。

下面的程序是字段接入性能测试的一个例子，包括基本的测试方法。每种方法测试字段接入的一种形式 -- accessSame 与同一对象的成员字段协作，accessOther 使用可直接接入的另一对象的字段，accessReflection 使用可通过反射接入的另一对象的字段。在每种情况下，方法执行相同的计算 -- 循环中简单的加/乘顺序。

程序如下：

```
public int accessSame(int loops) {

m_value = 0;

for (int index = 0; index < loops; index++) {

m_value = (m_value + ADDITIVE_VALUE) *

MULTIPLIER_VALUE;

}

return m_value;

}

public int acces

sReference(int loops) {

TimingClass timing = new TimingClass();

for (int index = 0; index < loops; index++) {

timing.m_value = (timing.m_value + ADDITIVE_VALUE) *

MULTIPLIER_VALUE;

}

return timing.m_value;
```

```
}

public int accessReflection(int loops) throws Exception {

TimingClass timing = new TimingClass();

try {

Field field = TimingClass.class.

getDeclaredField("m_value");

for (int index = 0; index < loops; index++) {

int value = (field.getInt(timing) +

ADDITIVE_VALUE) * MULTIPLIER_VALUE;

field.setInt(timing, value);

}

return timing.m_value;

} catch (Exception ex) {

System.out.println("Error using reflection");

throw ex;

}

}
```

在上面的例子中，测试程序重复调用每种方法，使用一个大循环数，从而平均多次调用的时间衡量结果。平均值中不包括每种方法第一次调用的时间，因此初始化时间不是结果中的一个因素。下面的图清楚的向我们展示了每种方法字段接入的时间：

图 1：字段接入时间 ：

我们可以看出：在前两副图中(Sun JVM)，使用反射的执行时间超过使用直接接入的 1000 倍以上。通过比较，IBM JVM 可能稍好一些，但反射方法仍旧需要比其它方法长 700 倍以上的时间。任何JVM 上其它两种方法之间时间方面无任何显著差异，但IBM JVM 几乎比 Sun JVM 快一倍。最有可能的是这种差异反映了 Sun Hot Spot JVM 的专业优化，它在简单基准

方面表现得很糟糕。反射性能是 Sun 开发 1.4 JVM 时关注的一个方面，它在反射方法调用结果中显示。在这类操作的性能方面，Sun 1.4.1 JVM 显示了比 1.3.1 版本很大的改进。

如果为为创建使用反射的对象编写了类似的计时测试程序，我们会发现这种情况下的差异不象字段和方法调用情况下那么显著。使用newInstance()调 用创建一个简单的java.lang.Object实例耗用的时间大约是在 Sun 1.3.1 JVM 上使用 new Object()的 12 倍，是在 IBM 1.4.0 JVM 的四倍，只是 Sun 1.4.1 JVM 上的两部。使用 Array.newInstance(type, size)创建一个数组耗用的时间是任何测试的 JVM 上使用 new type[size]的两倍，随着数组大小的增加，差异逐步缩小。随着 jdk6.0 的推出，反射机制的性能也有了很大的提升。期待中….

总结：

Java 语言反射提供一种动态链接程序组件的多功能方法。它允许程序创建和控制任何类的对象(根据安全性限制)，无需提前硬编码目标类。这些特性使得反射 特别适用于创建以非常普通的方式与对象协作的库。例如，反射经常在持续存储对象为数据库、XML 或其它外部格式的框架中使用。Java reflection 非常有用，它使类和数据结构能按名称动态检索相关信息，并允许在运行着的程序中操作这些信息。Java 的这一特性非常强大，并且是其它一些常用语言，如 C、C++、Fortran 或者 Pascal 等都不具备的。

但反射有两个缺点。第一个是性能问题。用于字段和方法接入时反射要远慢于直接代码。性能问题的程度取决于程序中是如何使用反射的。如果它作为程序运行中相 对很少涉及的部分，缓慢的性能将不会是一个问题。即使测试中最坏情况下的计时图显示的反射操作只耗用几微秒。仅反射在性能关键的应用的核心逻辑中使用时性 能问题才变得至关重要。

许多应用中更严重的一个缺点是使用反射会模糊程序内部实际要发生的事情。程序人员希望在源代码中看到程序的逻辑，反射等绕过了源代码的技术会带来维护问 题。反射代码比相应的直接代码更复杂，正如性能比较的代码实例中看到的一样。解决这些问题的最佳方案是保守地使用反射——仅在它可以真正增加灵活性的地方 ——记录其在目标类中的使用。

一下是对应各个部分的例子：

具体的应用：

1、 模仿 instanceof 运算符号

```
class A {}
```

```
public class instance1 {

public static  void main(String args[])

{

try {

Class cls = Class.forName("A");

boolean b1

= cls.isInstance(new Integer(37));

System.out.println(b1);

boolean b2 = cls.isInstance(new A());

System.out.println(b2);

}

catch (Throwable e) {

System.err.println(e);

}

}

}
```

2、 在类中寻找指定的方法，同时获取该方法的参数列表，例外和返回值

```
import java.lang.reflect.*;

public class method1 {

private int f1(

Object p, int x) throws NullPointerException

{
```

```
if (p == null)
throw new NullPointerException();
return x;
}
public static void main(String args[])
{
try {
Class cls = Class.forName("method1");
Method methlist[]
= cls.getDeclaredMethods();
for (int i = 0; i < methlist.length;
i++)
Method m = methlist[i];
System.out.println("name
= " + m.getName());
System.out.println("decl class = " +
m.getDeclaringClass());
Class pvec[] = m.getParameterTypes();
for (int j = 0; j < pvec.length; j++)
System.out.println("
param #" + j + " " + pvec[j]);
Class evec[] = m.getExceptionTypes();
```

```
for (int j = 0; j < evec.length; j++)
System.out.println("exc #" + j
+ " " + evec[j]);
System.out.println("return type = " +
m.getReturnType());
System.out.println("-----");
}
}
catch (Throwable e) {
System.err.println(e);
}
}
}
```

3、 获取类的构造函数信息，基本上与获取方法的方式相同

```
import java.lang.reflect.*;
public class constructor1 {
public constructor1()
{
}
protected constructor1(int i, double d)
{
}
```

```
public static void main(String args[])
{
try {
Class cls = Class.forName("constructor1");
Constructor ctorlist[]
= cls.getDeclaredConstructors();
for (int i = 0; i < ctorlist.length; i++) {
Constructor ct = ctorlist[i];
System.out.println("name
= " + ct.getName());
System.out.println("dec1 class = " +
ct.getDeclaringClass());
Class pvec[] = ct.getParameterTypes();
for (int j = 0; j < pvec.length; j++)
System.out.println("param #"
+ j + " " + pvec[j]);
Class evec[] = ct.getExceptionTypes();
for (int j = 0; j < evec.length; j++)
System.out.println(
"exc #" + j + " " + evec[j]);
System.out.println("-----");
}
```

```
}
catch (Throwable e) {
System.err.println(e);
}
}
}
```

4、 获取类中的各个数据成员对象，包括名称。类型和访问修饰符号

```
import java.lang.reflect.*;
public class field1 {
private double d;
public static final int i = 37;
String s = "testing";
public static void main(String args[])
{
try {
Class cls = Class.forName("field1");
Field fieldlist[]
= cls.getDeclaredFields();
for (int i
= 0; i < fieldlist.length; i++) {
Field fld = fieldlist[i];
System.out.println("name
```

```
= " + fld.getName());
System.out.println("decl class = " +
fld.getDeclaringClass());
System.out.println("type
= " + fld.getType());
int mod = fld.getModifiers();
System.out.println("modifiers = " +
Modifier.toString(mod));
System.out.println("-----");
}
}
catch (Throwable e) {
System.err.println(e);
}
}
}
```

5、 通过使用方法的名字调用方法

```
import java.lang.reflect.*;
public class method2 {
public int add(int a, int b)
{
return a + b;
```

```
}

public static  void main(String args[])

{

try {

Class cls = Class.forName("method2");

Class partypes[] = new Class[2];

partypes[0] = Integer.TYPE;

partypes[1] = Integer.TYPE;

Method meth = cls.getMethod(

"add", partypes);

method2 methobj = new method2();

Object arglist[] = new Object[2];

arglist[0] = new Integer(37);

arglist[1] = new Integer(47);

Object retobj

= meth.invoke(methobj, arglist);

Integer retval = (Integer)retobj;

System.out.println(retval.intValue());

}

catch (Throwable e) {

System.err.println(e);

}
```

```
}

}
```

6、 创建新的对象

```
import java.lang.reflect.*;

public class constructor2 {

public constructor2()

{

}

public constructor2(int a, int b)

{

System.out.println(

"a = " + a + " b = " + b);

}

public static  void main(String args[])

{

try {

Class cls = Class.forName("constructor2");

Class partypes[] = new Class[2];

partypes[0] = Integer.TYPE;

partypes[1] = Integer.TYPE;

Constructor ct

= cls.getConstructor(partypes);
```

```
Object arglist[] = new Object[2];

arglist[0] = new Integer(37);

arglist[1] = new Integer(47);

Object retobj = ct.newInstance(arglist);

}

catch (Throwable e) {

System.err.println(e);

}

}

}
```

7、 变更类实例中的数据的值

```
import java.lang.reflect.*;

public class field2 {

public double d;

public static  void main(String args[])

{

try {

Class cls = Class.forName("field2");

Field fld = cls.getField("d");

field2 f2obj = new field2();

System.out.println("d = " + f2obj.d);

fld.setDouble(f2obj, 12.34);
```

```
System.out.println("d = " + f2obj.d);

}

catch (Throwable e) {

System.err.println(e);

}

}

}
```

使用反射创建可重用代码：

1、 对象工厂

```
Object factory(String p) {

Class c;

Object o=null;

try {

c = Class.forName(p);// get class def

o = c.newInstance(); // make a new one

} catch (Exception e) {

System.err.println("Can't make a " + p);

}

return o;

}

public class ObjectFoundry {

public static  Object factory(String p)
```

```
throws ClassNotFoundException,

InstantiationException,

IllegalAccessException {

Class c = Class.forName(p);

Object o = c.newInstance();

return o;

}

}
```

2、 动态检测对象的身份，替代 instanceof

```
public static boolean

isKindOf(Object obj, String type)

throws ClassNotFoundException {

// get the class def for obj and type

Class c = obj.getClass();

Class tClass = Class.forName(type);

while ( c!=null ) {

if ( c==tClass ) return true;

c = c.getSuperclass();

}

return false;
```

Java 集合浅析

Vector 和 ArrayList 的区别

Vector 和 ArrayList 在使用上非常相似,都可用来表示一组数量可变的对象应用的集合,并且可以随机地访问其中的元素。
Vector 的方法都是同步的(Synchronizæd),是线程安全的(thread-safe),而 ArrayList 的方法不是,由于线程的同步必然要影响性能,因此,ArrayList 的性能比 Vector 好。
当 Vector 或 ArrayList 中的元素超过它的初始大小时,Vector 会将它的容量翻倍,而 ArrayList 只增加50%的大小,这样,ArrayList 就有利于节约内存空间。

Hashtable 和 HashMap 区别

Hashtable 和 HashMap 它们的性能方面的比较类似 Vector 和 ArrayList,比如 Hashtable 的方法是同步的,而 HashMap 的不是。

ArrayList 和 LinkedList 区别

对于处理一列数据项,Java 提供了两个类 ArrayList 和 LinkedList,ArrayList 的内部实现是基于内部数组 Object[],所 以从概念上讲,它更象数组,但 LinkedList 的内部实现是基于一组连接的记录,所以,它更象一个链表结构,所以,它们在性能上有很大的差别。
从上面的分析可知,在 ArrayList 的前面或中间插入数据时,你必须将其后的所有数据相应的后移,这样必然要花费较多时间,所以,当你的操作是在一列 数据的后面添加数据而不是在前面或中间,并且需要随机地访问其中的元素时,使用 ArrayList 会提供比较好的性能
而访问链表中的某个元素时,就必须从链表的一端开始沿着连接方向一个一个元素地去查找,直到找到所需的元素为止,所以,当你的操作是在一列数据的前面或中间添加或删除数据,并且按照顺序访问其中的元素时,就应该使用 LinkedList 了。
如果在编程中,1,2两种情形交替出现,这时,你可以考虑使用 List 这样的通用接口,而不用关心具体的实现,在具体的情形下,它的性能由具体的实现来保证。

配置集合类的初始大小

在 Java 集合框架中的大部分类的大小是可以随着元素个数的增加而相应的增加的,我们似乎不用关心它的初始大小,但如果我们考虑类的性能问题时,就一定要考虑尽可能地设置好集合对象的初始大小,这将大大提高代码的性能。
比如,Hashtable 缺省的初始大小为101,载入因子为0.75,即如果其中的元素个数超过75个,它就必须增加大小并重新组织元素,所以,如果你 知道在创建一个新的 Hashtable 对象时就知道

元素的确切数目如为110,那么,就应将其初始大小设为110/0.75=148,这样,就可以避免重 新组织内存并增加大小。

线性表，链表，哈希表是常用的数据结构，在进行 Java 开发时，JDK 已经为我们提供了一系列相应的类来实现基本的数据结构。这些类均在 java.util 包中。本文试图通过简单的描述，向读者阐述各个类的作用以及如何正确使用这些类。

Collection
├List
│ ├LinkedList
│ ├ArrayList
│ └Vector
│ └Stack
└Set
Map
├Hashtable
├HashMap
└WeakHashMap

Collection 接口

Collection 是最基本的集合接口，一个 Collection 代表一组 Object，即 Collection 的元素（Elements）。一些 Collection 允许相同的元素而另一些不行。一些能排序而另一些不行。Java SDK 不提供直接继承自 Collection 的类，Java SDK 提供的类都是继承自 Collection 的“子接口”如 List 和 Set。

所有实现 Collection 接口的类都必须提供两个标准的构造函数：无参数的构造函数用于创建一个空的 Collection，有一个 Collection 参数的构造函数用于创建一个新的 Collection，这个新的 Collection 与传入的 Collection 有相同的元素。后 一个构造函数允许用户复制一个 Collection。

如何遍历 Collection 中的每一个元素？不论 Collection 的实际类型如何，它都支持一个 iterator()的方法，该方法返回一个迭代子，使用该迭代子即可逐一访问 Collection 中每一个元素。典型的用法如下：

```
Iterator it = collection.iterator(); // 获得一个迭代子
while(it.hasNext()) {
    Object obj = it.next(); // 得到下一个元素
}
```

由 Collection 接口派生的两个接口是 List 和 Set。

List 接口

List 是有序的 Collection，使用此接口能够精确的控制每个元素插入的位置。用户能够

使用索引（元素在 List 中的位置，类似于数组下标）来访问 List 中的元素，这类似于 Java 的数组。

和下面要提到的 Set 不同，List 允许有相同的元素。

除了具有 Collection 接口必备的 iterator()方法外，List 还提供一个 listIterator()方法，返回一个 ListIterator 接口，和标准的 Iterator 接口相比，ListIterator 多了一些 add()之类的方法，允许添加，删除，设定元素， 还能向前或向后遍历。

实现 List 接口的常用类有 LinkedList，ArrayList，Vector 和 Stack。

LinkedList 类

LinkedList 实现了 List 接口，允许 null 元素。此外 LinkedList 提供额外的 get，remove，insert 方法在 LinkedList 的首部或尾部。这些操作使 LinkedList 可被用作堆栈（stack），队列（queue）或双向队列（deque）。

注意 LinkedList 没有同步方法。如果多个线程同时访问一个 List，则必须自己实现访问同步。一种解决方法是在创建 List 时构造一个同步的 List：

```
List list = Collections.synchronizedList(new LinkedList(...));
```

ArrayList 类

ArrayList 实现了可变大小的数组。它允许所有元素，包括 null。ArrayList 没有同步。size，isEmpty，get，set 方法运行时间为常数。但是 add 方法开销为分摊的常数，添加 n 个元素需要 O(n)的时间。其他的方法运行时间为线性。

每个 ArrayList 实例都有一个容量（Capacity），即用于存储元素的数组的大小。这个容量可随着不断添加新元素而自动增加，但是增长算法 并没有定义。当需要插入大量元素时，在插入前可以调用 ensureCapacity 方法来增加 ArrayList 的容量以提高插入效率。

和 LinkedList 一样，ArrayList 也是非同步的（unsynchronized）。

Vector 类

Vector 非常类似 ArrayList，但是 Vector 是同步的。由 Vector 创建的 Iterator，虽然和 ArrayList 创建的 Iterator 是同一接口，但是，因为 Vector 是同步的，当一个 Iterator 被创建

而且正在被使用，另一个线程改变了 Vector 的状态（例 如，添加或删除了一些元素），这时调用 Iterator 的方法时将抛出 ConcurrentModificationException，因此必须捕获该 异常。

Stack 类

Stack 继承自 Vector，实现一个后进先出的堆栈。Stack 提供5个额外的方法使得 Vector 得以被当作堆栈使用。基本的 push 和 pop 方法，还有 peek 方法得到栈顶的元素，empty 方法测试堆栈是否为空，search 方法检测一个元素在堆栈中的位置。Stack 刚创建后是空栈。

Set 接口

Set 是一种不包含重复的元素的 Collection，即任意的两个元素 e1 和 e2 都有 e1.equals(e2)=false，Set 最多有一个 null 元素。

很明显，Set 的构造函数有一个约束条件，传入的 Collection 参数不能包含重复的元素。

请注意：必须小心操作可变对象（Mutable Object）。如果一个 Set 中的可变元素改变了自身状态导致 Object.equals(Object)=true 将导致一些问题。

Map 接口

请注意，Map 没有继承 Collection 接口，Map 提供 key 到 value 的映射。一个 Map 中不能包含相同的 key，每个 key 只能映射一个 value。Map 接口提供3种集合的视图，Map 的内容可以被当作一组 key 集合，一组 value 集合，或者一组 key-value 映射。

Hashtable 类

Hashtable 继承 Map 接口，实现一个 key-value 映射的哈希表。任何非空（non-null）的对象都可作为 key 或者 value。

添加数据使用 put(key, value)，取出数据使用 get(key)，这两个基本操作的时间开销为常数。

Hashtable 通过 initial capacity 和 load factor 两个参数调整性能。通常缺省的 load factor 0.75 较好地实现了时间和空间的均衡。增大 load factor 可以节省空间但相应的查找时间将增大，这会影响像 get 和 put 这样的操作。

使用 Hashtable 的简单示例如下，将 1，2，3 放到 Hashtable 中，他们的 key 分别是”one”，”two”，”three”：

```
Hashtable numbers = new Hashtable();
numbers.put(“one”, new Integer(1));
numbers.put(“two”, new Integer(2));
numbers.put(“three”, new Integer(3));
```

要取出一个数，比如2，用相应的 key：

```
Integer n = (Integer)numbers.get("two");
System.out.println("two = " + n);
```

由于作为 key 的对象将通过计算其散列函数来确定与之对应的 value 的位置，因此任何作为 key 的对象都必须实现 hashCode 和 equals 方 法。hashCode 和 equals 方法继承自根类 Object，如果你用自定义的类当作 key 的话，要相当小心，按照散列函数的定义，如果两个对象相 同，即 obj1.equals(obj2)=true，则它们的 hashCode 必须相同，但如果两个对象不同，则它们的 hashCode 不一定不同，如 果两个不同对象的 hashCode 相同，这种现象称为冲突，冲突会导致操作哈希表的时间开销增大，所以尽量定义好的 hashCode()方法，能加快哈希 表的操作。

如果相同的对象有不同的 hashCode，对哈希表的操作会出现意想不到的结果（期待的 get 方法返回 null），要避免这种问题，只需要牢记一条：要同时复写 equals 方法和 hashCode 方法，而不要只写其中一个。

Hashtable 是同步的。

HashMap 类

HashMap 和 Hashtable 类似，不同之处在于 HashMap 是非同步的，并且允许 null，即 null value 和 null key。，但是将 HashMap 视为 Collection 时（values()方法可返回 Collection），其迭代子操作时间开销和 HashMap 的容量成比例。因此，如果迭代操作的性能相当重要的话，不要将 HashMap 的初始化容量设得过高，或者 load factor 过低。

WeakHashMap 类

WeakHashMap 是一种改进的 HashMap，它对 key 实行"弱引用"，如果一个 key 不再被外部所引用，那么该 key 可以被 GC 回收。

总结

如果涉及到堆栈，队列等操作，应该考虑用 List，对于需要快速插入，删除元素，应该使用 LinkedList，如果需要快速随机访问元素，应该使用 ArrayList。

如果程序在单线程环境中，或者访问仅仅在一个线程中进行，考虑非同步的类，其效率较高，如果多个线程可能同时操作一个类，应该使用同步的类。

要特别注意对哈希表的操作，作为 key 的对象要正确复写 equals 和 hashCode 方法。

尽量返回接口而非实际的类型，如返回 List 而非 ArrayList，这样如果以后需要将 ArrayList 换成 LinkedList 时，客户端代码不用改变。这就是针对抽象编程。

java 的类反射机制

什么是反射：

反射的概念是由 Smith 在 1982 年首次提出的，主要是指程序可以访问、检测和修改它本身状态或行为的一种能力。这一概念的提出很快引发了计算机科学领 域关于应用反射性的研究。它首先被程序语言的设计领域所采用,并在 Lisp 和面向对象方面取得了成绩。其中 LEAD/LEAD++ 、OpenC++ 、MetaXa 和 OpenJava 等就是基于反射机制的语言。最近，反射机制也被应用到了视窗系统、操作系统和文件系统中。

反射本身并不是一个新概念，尽管计算机科学赋予了反射概念新的含义。在计算机科学领域，反射是指一类应用，它们能够自描述和自控制。也就是说，这类应用通 过采用某种机制来实现对自己行为的描述（self-representation）和监测（examination），并能根据自身行为的状态和结果，调 整或修改应用所描述行为的状态和相关的语义。

二、什么是 Java 中的类反射：

Reflection 是 Java 程序开发语言的特征之一，它允许运行中的 Java 程序对自身进行检查，或者说“自审”，并能直接操作程序的内部属性和方法。 Java 的这一能力在实际应用中用得不是很多，但是在其它的程序设计语言中根本就不存在这一特性。例如，Pascal、C 或者 C++ 中就没有办法在程序中获得函数定义相关的信息。
Reflection 是 Java 被视为动态（或准动态）语言的关键，允许程序于执行期 Reflection APIs 取得任何已知名称之 class 的内部信息，包括 package、type parameters、superclass、implemented interfaces、inner classes, outer class, fields、constructors、methods、modifiers，並可于执行期生成 instances、变更 fields 内容或唤起 methods。

三、Java 类反射中所必须的类：

Java 的类反射所需要的类并不多，它们分别是：Field、Constructor、Method、Class、Object，下面我将对这些类做一个简单的说明。
Field 类：提供有关类或接口的属性的信息，以及对它的动态访问权限。反射的字段可能是一个类（静态）属性或实例属性，简单的理解可以把它看成一个封装反射类的属性的类。

Constructor 类：提供关于类的单个构造方法的信息以及对它的访问权限。这个类和 Field 类不同，Field 类封装了反射类的属性，而 Constructor 类则封装了反射类的构造方法。
Method 类：提供关于类或接口上单独某个方法的信息。所反映的方法可能是类方法或实例方法（包括抽象方法）。 这个类不难理解，它是用来封装反射类方法的一个类。
Class 类：类的实例表示正在运行的 Java 应用程序中的类和接口。枚举是一种类，注释是一种接口。每个数组属于被映射为 Class 对象的一个类，所有具有相同元素类型和维数的数组都共享该 Class 对象。
Object 类：每个类都使用 Object 作为超类。所有对象（包括数组）都实现这个类的方法。

四、Java 的反射类能做什么：

看完上面的这么多我想你已经不耐烦了，你以为我在浪费你的时间，那么好吧！下面我们就用一些简单的小例子来说明它。
首先我们来看一下通过 Java 的反射机制我们能得到些什么。
首先我们来写一个类：

java 代码

```
1  import java.awt.event.ActionListener;
2  import java.awt.event.ActionEvent;
3  class A extends Object implements ActionListener{
4  private int a = 3;
5  public Integer b = new Integer(4);
6  public A(){}
7  public A(int id,String name){}
8  public int abc(int id,String name){return 0;}
9  public void actionPerformed(ActionEvent e){}
10 }
```

你可能被我这个类弄糊涂了，你看不出我要做什么，那就不要看这个类了，这个类是用来测试的，你知道知道它继承了 Object 类，有一个接口是 ActionListener，两个属性 int 和 Integer, 两个构造方法和两个方法，这就足够了。
下面我们把 A 这个类作为一个反射类，来过去 A 类中的一些信息，首先我们先来过去一下反射类中的属性和属性值。

java 代码

```
11 import java.lang.reflect.*;
12 class B{
13 public static void main(String args[]){
14 A r = new A();
15 Class temp = r.getClass();
16 try{
17 System.out.println("反射类中所有公有的属性");
18 Field[] fb =temp.getFields();
19 for(int j=0;j<fb.length;j++){
20 Class cl= fb[j].getType();
21 System.out.println("fb:"+cl);
22 }
23
24 System.out.println("反射类中所有的属性");
25 Field[] fa = temp.getDeclaredFields();
26 for(int j=0;j<fa.length;j++){
27 Class cl= fa[j].getType();
28 System.out.println("fa:"+cl);
29 }
30 System.out.println("反射类中私有属性的值");
31 Field f = temp.getDeclaredField("a");
32 f.setAccessible(true);
33 Integer i = (Integer)f.get(r);
34 System.out.println(i);
35 }catch(Exception e){
36 e.printStackTrace();
37 }
38 }
39
40 }
```

这里用到了两个方法，getFields()、getDeclaredFields()，它们分别是用来获取反射类中所有公有属性和反射类中所有的属性的 方法。另外还有 getField(String) 和 getDeclaredField(String)方法都是用来过去反射类中指定的属性的方法，要注意 的是 getField 方法只能取到反射类中公有的属性，而 getDeclaredField 方法都能取到。
这里还用到了 Field 类的 setAccessible 方法，它是用来设置是否有权限访问反射类中的私有属性的，只有设置为 true 时才可以访问，默认为 false。另外 Field 类还有 set(Object AttributeName,Object value)方法，可以改变指定属性的值。

下面我们来看一下如何获取反射类中的构造方法

java 代码

```
41 import java.lang.reflect.*;
42 public class SampleConstructor {
43 public static void main(String[] args) {
44 A r = new A();
45 printConstructors(r);
46 }
47
48 public static void printConstructors(A r) {
49 Class c = r.getClass();
50 //获取指定类的类名
51 String className = c.getName();
52 try {
53 //获取指定类的构造方法
54 Constructor[] theConstructors = c.getConstructors();
55 for(int i=0; i<theConstructors.length; i++) {
56 //获取指定构造方法的参数的集合
57 Class[] parameterTypes = theConstructors[i].getParameterTypes();
58
59 System.out.print(className + "(");
60
61 for(int j=0; j<parameterTypes.length; j++)
62 System.out.print(parameterTypes[j].getName() + " ");
63
64 System.out.println(")");
65
66 }
67 }catch(Exception e) {
68 e.printStackTrace();
69 }
70 }
71 }
```

这个例子很简单,只是用 getConstructors()方法获取了反射类的构造方法的集合，并用 Constructor 类的 getParameterTypes()获取该构造方法的参数。

下面我们再来获取一下反射类的父类（超类）和接口

java 代码

```
72 import java.io.*;
73 import java.lang.reflect.*;
74
75 public class SampleInterface {
```

```
76  public static void main(String[] args) throws Exception {
77  A raf = new A();
78  printInterfaceNames(raf);
79  }
80
81  public static void printInterfaceNames(Object o) {
82  Class c = o.getClass();
83  //获取反射类的接口
84  Class[] theInterfaces = c.getInterfaces();
85  for(int i=0; i<theInterfaces.length; i++)
86  System.out.println(theInterfaces[i].getName());
87  //获取反射类的父类（超类）
88  Class theSuperclass = c.getSuperclass();
89  System.out.println(theSuperclass.getName());
90  }
91  }
```

这个例子也很简单，只是用 Class 类的 getInterfaces()方法获取反射类的所有接口，由于接口可以有多个，所以它返回一个 Class 数组。用 getSuperclass()方法来获取反射类的父类（超类），由于一个类只能继承自一个类，所以它返回一个 Class 对象。

下面我们来获取一下反射类的方法

java 代码

```
92  import java.lang.reflect.*;
93  public class SampleMethod {
94
95  public static void main(String[] args) {
96  A p = new A();
97  printMethods(p);
98  }
99
100 public static void printMethods(Object o) {
101 Class c = o.getClass();
102 String className = c.getName();
103 Method[] m = c.getMethods();
104 for(int i=0; i<m.length; i++) {
105 //输出方法的返回类型
106 System.out.print(m[i].getReturnType().getName());
107 //输出方法名
108 System.out.print(" "+m[i].getName()+"(");
```

```
109 //获取方法的参数
110 Class[] parameterTypes = m[i].getParameterTypes();
111 for(int j=0; j<parameterTypes.length; j++){
112 System.out.print(parameterTypes[j].getName());
113 if(parameterTypes.length>j+1){
114 System.out.print(",");
115 }
116 }
117
118 System.out.println(")");
119 }
120
121 }
122
123 }
```

这个例子并不难，它只是获得了反射类的所有方法，包括继承自它父类的方法。然后获取方法的返回类型、方法名和方法参数。

接下来让我们回过头来想一想，我们获取了反射类的属性、构造方法、父类、接口和方法，可这些东西能帮我们做些什么呢！！
下面我写一个比较完整的小例子，来说明 Java 的反射类能做些什么吧！！

java 代码

```
124 import java.lang.reflect.Constructor;
125 import java.lang.reflect.Method;
126
127 public class LoadMethod {
128 public Object Load(String cName,String MethodName,String[] type,String[] param){
129 Object retobj = null;
130 try {
131 //加载指定的 Java 类
132 Class cls = Class.forName(cName);
133
134 //获取指定对象的实例
135 Constructor ct = cls.getConstructor(null);
136 Object obj = ct.newInstance(null);
137
138 //构建方法参数的数据类型
139 Class partypes[] = this.getMethodClass(type);
140
```

```
141 //在指定类中获取指定的方法
142 Method meth = cls.getMethod(MethodName, partypes);
143
144 //构建方法的参数值
145 Object arglist[] = this.getMethodObject(type,param);
146
147 //调用指定的方法并获取返回值为 Object 类型
148 retobj= meth.invoke(obj, arglist);
149
150 }
151 catch (Throwable e) {
152 System.err.println(e);
153 }
154 return retobj;
155 }
156
157 //获取参数类型 Class[]的方法
158 public Class[] getMethodClass(String[] type){
159 Class[] cs = new Class[type.length];
160 for (int i = 0; i < cs.length; i++) {
161 if(!type[i].trim().equals("")||type[i]!=null){
162 if(type[i].equals("int")||type[i].equals("Integer")){
163 cs[i]=Integer.TYPE;
164 }else if(type[i].equals("float")||type[i].equals("Float")){
165 cs[i]=Float.TYPE;
166 }else if(type[i].equals("double")||type[i].equals("Double")){
167 cs[i]=Double.TYPE;
168 }else if(type[i].equals("boolean")||type[i].equals("Boolean")){
169 cs[i]=Boolean.TYPE;
170 }else{
171 cs[i]=String.class;
172 }
173 }
174 }
175 return cs;
176 }
177
178 //获取参数 Object[]的方法
179 public Object[] getMethodObject(String[] type,String[] param){
180 Object[] obj = new Object[param.length];
181 for (int i = 0; i < obj.length; i++) {
182 if(!param[i].trim().equals("")||param[i]!=null){
183 if(type[i].equals("int")||type[i].equals("Integer")){
184 obj[i]= new Integer(param[i]);
```

```
185 }else if(type[i].equals("float")||type[i].equals("Float")){
186 obj[i]= new Float(param[i]);
187 }else if(type[i].equals("double")||type[i].equals("Double")){
188 obj[i]= new Double(param[i]);
189 }else if(type[i].equals("boolean")||type[i].equals("Boolean")){
190 obj[i]=new Boolean(param[i]);
191 }else{
192 obj[i] = param[i];
193 }
194 }
195 }
196 return obj;
197 }
198 }
```

这是我在工作中写的一个实现 Java 在运行时加载指定的类，并调用指定方法的一个小例子。这里没有 main 方法，你可以自己写一个。
Load 方法接收的五个参数分别是，Java 的类名，方法名，参数的类型和参数的值。

结束语：
Java 语言反射提供一种动态链接程序组件的多功能方法。它允许程序创建和控制任何类的对象，无需提前硬编码目标类。这些特性使得反射特别适用于创建以非 常普通的方式与对象协作的库。Java reflection 非常有用，它使类和数据结构能按名称动态检索相关信息，并允许在运行着的程序中操作这些信息。Java 的这一特性非常强大，并且是其它一些常用语言，如 C、C++、Fortran 或者 Pascal 等都不具备的。

但反射有两个缺点。第一个是性能问题。用于字段和方法接入时反射要远慢于直接代码。性能问题的程度取决于程序中是如何使用反射的。如果它作为程序运行中相 对很少涉及的部分，缓慢的性能将不会是一个问题。即使测试中最坏情况下的计时图显示的反射操作只耗用几微秒。仅反射在性能关键的应用的核心逻辑中使用时性 能问题才变得至关重要。

由于是第一次写博客，写的不好还请大家多多指导，当然你的鼓励是我生命的源泉。

JAVA 反射机制的学习

JAVA 语言中的反射机制：

在 Java 运行时 环境中，对于任意一个类，能否知道这个类有哪些属性和方法？

对于任意一个对象，能否调用他的方法？这些答案是肯定的，这种动态获取类的信息，以及动态调用类的方法的功能来源于 JAVA 的反射。从而使 java 具有动态语言的特性。

JAVA反射机制主要提供了以下功能：

1.在运行时判断任意一个对象所属的类

2.在运行时构造任意一个类的对象

3.在运行时判断任意一个类所具有的成员变量和方法（通过反射甚至可以调用 private 方法）

4.在运行时调用任意一个对象的方法（*****注意：前提都是在运行时，而不是在编译时）

Java 反射相关的 API 简介：

位于 java。lang。reflect 包中

--Class 类：代表一个类

--Filed 类：代表类的成员变量

--Method 类：代表类的方法

--Constructor 类：代表类的构造方法

--Array 类：提供了动态创建数组，以及访问数组的元素的静态方法。该类中的所有方法都是静态方法

---Class 类

在 java 的 Object 类中的申明了数个应该在所有的 java 类中被改写的 methods：hashCode(), equals(),clone(),toString(),getClass()等，其中的 getClass()返回 yige Class 类型的对象。

Class 类十分的特殊，它和一般的类一样继承自 Object，其实体用以表达 java 程序运行时的 class 和 interface，也用来表达 enum，array，primitive，Java Types 以及关键字 void，当加载一个类，或者当加载器（class loader）的 defineClass（）被 JVM 调用，便产生一个 Class 对象，

Class 是 Reflection 起源，针对任何你想探勘的 class（类），唯有现为他产生一个 Class 的对象，接下来才能经由后者唤起为数十多个的反射 API。

Java 允许我们从多种途径为一个类 class 生成对应的 Class 对象。

--运用 getClass（）：Object 类中的方法，每个类都拥有此方法

```
String str="abc";
Class cl=str.getClass();
```

--运用 Class。getSuperclass（）：Class 类中的方法，返回该 Class 的父类的 Class

--运用 Class。forName（）静态方法：

--运用 ,Class：类名.class

--运用 primitive wrapper classes 的 TYPE 语法： 基本类型包装类的 TYPE，如：Integer.TYPE

注意：TYPE 的使用，只适合原生（基本）数据类型

---运行时生成 instance

想生成对象的实体，在反射动态机制中有两种方法，一个针对无变量的构造方法，一个针对带参数的
构造方法，，如果想调用带参数的构造方法，就比较的麻烦，不能直接调用 Class 类中的 newInstance（）
，而是调用 Constructor 类中 newInstance（）方法，首先准备一个 Class[]作为 Constructor 的参数类型。
然后调用该 Class 对象的 getConstructor（）方法获得一个专属的 Constructor 的对象，最后再准备一个
Object[]作为 Constructor 对象昂的 newInstance（）方法的实参。

在这里需要说明的是 只有两个类拥有 newInstance（）方法，分别是 Class 类和 Constructor 类
Class 类中的 newInstance（）方法是不带参数的，而 Constructro 类中的 newInstance（）方法是带参数的
需要提供必要的参数。

例:

```
Class c=Class.forName("DynTest");
Class[] ptype=new Class[]{double.class,int.class};
Constructor ctor=c.getConstructor(ptypr);
Object[] obj=new Object[]{new Double(3.1415),new Integer(123)};
Object object=ctor.newInstance(obj);
System.out.println(object);
```

---运行时调用 Method

这个动作首先准备一个 Class[]{}作为 getMethod（String name，Class[]）方法的参数类型，接下来准备一个
Obeject[]放置自变量，然后调用 Method 对象的 invoke（Object obj，Object[]）方法。

注意，在这里调用

---运行时调用 Field 内容

变更 Field 不需要参数和自变量，首先调用 Class 的 getField（）并指定 field 名称，获得特定的 Field 对象后
便可以直接调用 Field 的 get（Object obj）和 set(Object obj,Object value)方法

java 代码

```
1   package cn.com.reflection;
2
3   import java.lang.reflect.Field;
4   import java.lang.reflect.InvocationTargetException;
5   import java.lang.reflect.Method;
6
7   public class ReflectTester {
8
9       /**
10       * 在这个类里面存在有 copy（）方法，根据指定的方法的参数去 构造一个新的对象的拷贝
11       * 并将他返回
12       * @throws NoSuchMethodException
13       * @throws InvocationTargetException
14       * @throws IllegalAccessException
15       * @throws InstantiationException
16       * @throws SecurityException
17       * @throws IllegalArgumentException
18       */
19      public Object copy(Object obj) throws IllegalArgumentException, SecurityException, InstantiationException, IllegalAccessException, InvocationTargetException, NoSuchMethodException{
20
21          //获得对象的类型
22          Class classType=obj.getClass();
23          System.out.println("该对象的类型是："+classType.toString());
24
25          //通过默认构造方法去创建一个新的对象， getConstructor 的视其参数决定调用哪个构造方法
26          Object objectCopy=classType.getConstructor(new Class[]{}).newInstance(new Object[]{});
27
28          //获得对象的所有属性
29          Field[] fields=classType.getDeclaredFields();
30
31          for(int i=0;i
32              //获取数组中对应的属性
33              Field field=fields[i];
34
35              String fieldName=field.getName();
36              String stringLetter=fieldName.substring(0, 1).toUpperCase();
```

```
37
38                  //获得相应属性的 getXXX 和 setXXX 方法名称
39                  String getName="get"+stringLetter+fieldName.substring(1);
40                  String setName="set"+stringLetter+fieldName.substring(1);
41
42                  //获取相应的方法
43                  Method getMethod=classType.getMethod(getName, new Class[]{});
44                              Method setMethod=classType.getMethod(setName, new
Class[]{field.getType()});
45
46                  //调用源对象的 getXXX（）方法
47                  Object value=getMethod.invoke(obj, new Object[]{});
48                  System.out.println(fieldName+" :"+value);
49
50                  //调用拷贝对象的 setXXX（）方法
51                  setMethod.invoke(objectCopy,new Object[]{value});
52
53
54              }
55
56              return objectCopy;
57
58          }
59
60
61              public static void main(String[] args) throws IllegalArgumentException,
SecurityException, InstantiationException, IllegalAccessException,
InvocationTargetException, NoSuchMethodException {
62              Customer customer=new Customer();
63              customer.setName("hejianjie");
64              customer.setId(new Long(1234));
65              customer.setAge(19);
66
67              Customer customer2=null;
68              customer2=(Customer)new ReflectTester().copy(customer);
69                          System.out.println(customer.getName()+"  "+customer2.getAge()+"
"+customer2.getId());
70
71              System.out.println(customer);
72              System.out.println(customer2);
73
74
75          }
76
```

```
77  }
78
79
80  class Customer{
81
82      private Long id;
83
84      private String name;
85
86      private int age;
87
88
89      public Customer(){
90
91      }
92
93      public int getAge() {
94          return age;
95      }
96
97
98      public void setAge(int age) {
99          this.age = age;
100     }
101
102
103     public Long getId() {
104         return id;
105     }
106
107
108     public void setId(Long id) {
109         this.id = id;
110     }
111
112
113     public String getName() {
114         return name;
115     }
116
117
118     public void setName(String name) {
119         this.name = name;
120     }
```

```
121
122 }
```

java 代码

```
123 package cn.com.reflection;
124
125 import java.lang.reflect.Array;
126
127 public class ArrayTester1 {
128
129     /**
130      * 此类根据反射来创建
131      * 一个动态的数组
132      */
133     public static void main(String[] args) throws ClassNotFoundException {
134
135         Class classType=Class.forName("java.lang.String");
136
137         Object array= Array.newInstance(classType,10);   //指定数组的类型和大小
138
139          //对索引为5的位置进行赋值
140         Array.set(array, 5, "hello");
141
142         String s=(String)Array.get(array, 5);
143
144         System.out.println(s);
145
146
147         //循环遍历这个动态数组
148         for(int i=0;i<((String[])array).length;i++){
149
150             String str=(String)Array.get(array, i);
151
152             System.out.println(str);
153         }
154
155     }
156
157 }
```

Java 反射 Reflection--运行时生成 instance

想生成对象的实体，在反射动态机制中有两种方法，一个针对无变量的构造方法，一个针对带参数的构造方法，，如果想调用无参数的构 造函数直接调用 Class 类中的 newInstance（），而如果想调用有参数的构造函数，则需要调用 Constructor 类中 newInstance（）方法，首先准备一个 Class[]作为 Constructor 的参数类型。然后调用该 Class 对象的 getConstructor（）方法获得一个专属的 Constructor 的对象，最后再准备一个 Object[]作为 Constructor 对象昂的 newInstance（）方法的实参。

在这里需要说明的是 只有两个类拥有 newInstance（）方法，分别是 Class 类和 Constructor 类，Class 类中的 newInstance（）方法是不带 参数的，而 Constructro 类中的 newInstance（）方法是带参数的需要提供必要的参数。

下面提供的代码是构造 Customer2类的三个构造函数

java 代码

```
1  package cn.com.reflection;
2
3  import java.lang.reflect.Constructor;
4  import java.lang.reflect.InvocationTargetException;
5
6  /**
7   * 在反射 Reflection 机制中，想生成一个类的实例有两种方法
8   * 一个是针对无参数的构造函数 ，另一个是针对有参数的构造函数
9   *
10  */
11 public class ReflecTest3 {
12
13      /**
14       * 反射的动态性质之一： 运行期动态生成 Instance
15       * @throws IllegalAccessException
16       * @throws InstantiationException
17       * @throws NoSuchMethodException
18       * @throws InvocationTargetException
19       * @throws SecurityException
20       * @throws IllegalArgumentException
21       */
22          public static void main(String[] args) throws InstantiationException,
IllegalAccessException, IllegalArgumentException, SecurityException,
InvocationTargetException, NoSuchMethodException {
```

```
23
24              Customer2 customer=new Customer2();
25
26              Class cls=customer.getClass();
27
28              //获得 Class 所代表的对象的所有类型的构造函数 Constructor 的数组
29              Constructor ctor[]=cls.getDeclaredConstructors();
30
31              for(int i=0;i
32                  //获得对应的构造函数参数列表的 Class 类型的数组
33                  Class cx[]=ctor[i].getParameterTypes();
34
35                  if(cx.length==0){
36                          //无参的构造函数可以通过 Class 实例直接调用 Class 类的
newInstance（）方法
37                      Object obj=cls.newInstance();
38                      //同样也可以象以下这样构造，调用 Constructor 类的 newInstance
（）方法
39                                                                  //Customer2
obj=(Customer2)cls.getConstructor(cx).newInstance(new Object[]{});
40                      System.out.println(obj);
41                  }else if(cx.length==2){
42     //此时只能调用 Constructor 类的 newInstance（）方法,注意：利用反射调用的是私
有 private 的构造方法
43                  Customer2 obj=(Customer2)cls.getConstructor(cx).newInstance(
44                                              new Object[]{new Long(123),"hejianjie"});
45                      System.out.println(obj);
46                  }else if(cx.length==3){
47     //此时只能调用 Constructor 类的 newInstance（）方法,注意：利用反射调用的是公
有 public 的 构 造 方 法
Customer2 obj=(Customer2)cls.getConstructor(cx).newInstance(
48                                          new Object[]{new Long(133),"China-Boy",new
Integer(21)});
49                      System.out.println(obj);
50                  }
51              }
52
53
54          }
55
56  }
57
58  class Customer2{
59
```

```
60      private Long id;
61
62      private String name;
63
64      private int age;
65
66      /**
67       * 无参数的构造函数
68       *
69       */
70      public Customer2(){
71
72      }
73
74      /**
75       * public 修饰的有参数的构造函数，3个参数
76       * @param id
77       * @param name
78       * @param age
79       */
80      public Customer2(Long id,String name,int age){
81          this.id=id;
82          this.name=name;
83          this.age=age;
84      }
85
86      /**
87       * public 修饰的构造函数,2个参数
88       * @param id
89       * @param name
90       */
91      public Customer2(Long id,String name){
92                  this.id=id;
93                  this.name=name;
94                  this.age=age;
95      }
96
97      public int getAge() {
98          return age;
99      }
100
101
102     public void setAge(int age) {
103         this.age = age;
```

```
104     }
105
106
107     public Long getId() {
108         return id;
109     }
110
111
112     public void setId(Long id) {
113         this.id = id;
114     }
115
116
117     public String getName() {
118         return name;
119     }
120
121
122     public void setName(String name) {
123         this.name = name;
124     }
125
126     public String toString(){
127                         return   ("id=="+this.getId()+"       Name=="+this.getName()+"
Age:"+this.getAge());
128     }
129
130 }
```

程序执行结果如下：

java 代码

```
131 id==123    Name==hejianjie Age:0
132 id==133    Name==China-Boy Age:21
133 id==null    Name==null Age:0
```

显然，第一行是执行的两个参数的构造函数，int 类型的 age 初始化为0，第二行执行的是带三个参数的构造函数，最后，第三行执行的是无参数的构造函数。，Long，String 类型的 id，name 都初始化为 null，而 int 类型的 age 初始化为0。

Java 反射机制

摘要

Reflection 是 Java 被视为动态（或准动态）语言的一个关键性质。这个机制允许程序在运行时透过 Reflection APIs 取得任何一个已知名称的 class 的内部信息，包括其 modifiers（诸如 public, static 等等）、superclass（例如 Object）、实现之 interfaces（例如 Cloneable），也包括 fields 和 methods 的所有 信息，并可于运行时改变 fields 内容或唤起 methods。本文借由实例，大面积示范 Reflection APIs。
关键词：
Introspection（内省、内观）Reflection（反射）

有时候我们说某个语言具有很强的动态性，有时候我们会区分动态和静态的不同技术与作法。我们朗朗上口动态绑定（dynamic binding）、动态链接（dynamic linking）、动态加载（dynamic loading）等。然而“动态”一词其实没有绝对而普遍适用的严格定义，有时候甚至像对象导向当初被导入编程领域一样，一人一把号，各吹各的调。
一般而言，开发者社群说到动态语言，大致认同的一个定义是：“程序运行时，允许改变程序结构或变量类型，这种语言称为动态语言”。从这个观点看， Perl，Python，Ruby 是动态语言，C++，Java，C#不是动态语言。尽管在这样的定义与分类下 Java 不是动态语言，它却有着一个非常突 出的动态相关机制：Reflection。这个字的意思是“反射、映象、倒影”，用在 Java 身上指的是我们可以于运行时加载、探知、使用编译期间完全未 知的 classes。换句话说，Java 程序可以加载一个运行时才得知名称的 class，获悉其完整构造（但不包括 methods 定义），并生成其对象实 体、或对其 fields 设值、或唤起其 methods1。这种“看透 class”的能力（the ability of the program to examine itself）被称为 introspection（内省、内观、反省）。Reflection 和 introspection 是常被并提的两个术语。
Java 如何能够做出上述的动态特性呢？这是一个深远话题，本文对此只简单介绍一些概念。整个篇幅最主要还是介绍 Reflection APIs，也就是让读者知道如何探索 class 的结构、如何对某个“运行时才获知名称的 class”生成一份实体、为其 fields 设值、调用其 methods。本文将谈到 java.lang.Class，以及 java.lang.reflect 中的 Method、Field、 Constructor 等等 classes。
“Class”class
众所周知 Java 有个 Object class，是所有 Java classes 的继承根源，其内声明了数个应该在所有 Java class 中被改写的 methods：hashCode()、equals()、clone()、toString()、getClass()等。其中 getClass()返回一个 Class object。
Class class 十分特殊。它和一般 classes 一样继承自 Object，其实体用以表达 Java 程序运行时的 classes 和 interfaces，也用来 表达 enum、array、primitive Java types（boolean, byte, char, short, int, long, float, double）以及关键词 void。当一个 class 被加载，或当加载器（class loader）的 defineClass()被 JVM 调用， JVM 便自动产生一个 Class object。如果您想借由“修改 Java 标准库源码”来观察 Class object 的实际生成时机（例如在 Class 的 constructor 内添加一个 println()），不能够！因为 Class 并没有 public constructor（见图 1）。本文最后我会拨一小块

篇幅顺带谈谈 Java 标准库源码的改动办法。Class 是 Reflection 故事起源。针对任 何您想探勘的 class，唯有先为它产生一个 Class object，接下来才能经由后者唤起为数十多个的 Reflection APIs。extends AbstractSequentialList.

.....
接下来讨论 Reflection 的另三个动态性质：(1) 运行时生成 instances，(2) 执行期唤起 methods，(3) 运行时改动 fields。
运行时生成 instances

欲生成对象实体，在 Reflection 动态机制中有两种作法，一个针对“无自变量 ctor”，

一个针对“带参数 ctor”。图 6 是面对“无自变量 ctor”的例子。如果欲调用的是“带参数 ctor“就比较麻烦些，图 7 是个例子，其中不再调用 Class 的 newInstance()，而是调用 Constructor 的 newInstance()。图 7 首先准备一个 Class[]做为 ctor 的参数类型（本例指定为一个 double 和一个 int），然后以此为自变量调 用 getConstructor()，获得一个专属 ctor。接下来再准备一个 Object[] 做为 ctor 实参值（本例指定 3.14159 和 125），调用上述专属 ctor 的 newInstance()。

```
#001 Class c = Class.forName("DynTest");

#002 Object obj = null;

#003 obj = c.newInstance(); //不带自变量

#004 System.out.println(obj);
```

图 6：动态生成“Class object 所对应之 class”的对象实体；无自变量。

```
#001 Class c = Class.forName("DynTest");

#002 Class[] pTypes = new Class[] { double.class, int.class };

#003 Constructor ctor = c.getConstructor(pTypes);

#004 //指定 parameter list，便可获得特定之 ctor

#005

#006 Object obj = null;

#007 Object[] arg = new Object[] {3.14159, 125}; //自变量
```

```
#008 obj = ctor.newInstance(arg);

#009 System.out.println(obj);
```

图 7：动态生成“Class object 对应之 class”的对象实体；自变量以 Object[]表示。

运行时调用 methods

这个动作和上述调用“带参数之 ctor”相当类似。首先准备一个 Class[]做为 ctor 的参数类型（本例指定其中一个是 String，另一个是 Hashtable），然后以此为自变量调用 getMethod()，获得特定的 Method object。接下来准备一个 Object[]放置自变量，然后调用上述所得之特定 Method object 的 invoke()，如图 8。知道为什么索取 Method object 时不需指定回返类型吗？因为 method overloading 机制要求 signature（署名式）必须唯一，而回返类型并非 signature 的一个成份。换句话说，只要指定了 method 名称和参数列，就一定指出了一个独一无二的 method。

```
#001 public String func(String s, Hashtable ht)

#002 {

#003 …System.out.println("func invoked"); return s;

#004 }

#005 public static void main(String args[])

#006 {

#007 Class c = Class.forName("Test");

#008 Class ptypes[] = new Class[2];

#009 ptypes[0] = Class.forName("java.lang.String");

#010 ptypes[1] = Class.forName("java.util.Hashtable");

#011 Method m = c.getMethod("func",ptypes);

#012 Test obj = new Test();
```

```
#013 Object args[] = new Object[2];

#014 arg[0] = new String("He llo,world");

#015 arg[1] = null;

#016 Object r = m.invoke(obj, arg);

#017 Integer rval = (String)r;

#018 System.out.println(rval);

#019 }
```

图 8：动态唤起 method

运行时变更 fields 内容

与先前两个动作相比，“变更 field 内容”轻松多了，因为它不需要参数和自变量。首先调用 Class 的 getField()并指定 field 名称。获得特定的 Field object 之后便可直接调用 Field 的 get() 和 set()，如图 9。

```
#001 public class Test {

#002 public double d;

#003

#004 public static void main(String args[])

#005 {

#006 Class c = Class.forName("Test");

#007 Field f = c.getField("d"); //指定 field 名称

#008 Test obj = new Test();

#009 System.out.println("d= " + (Double)f.get(obj));
```

```
#010 f.set(obj, 12.34);

#011 System.out.println("d= " + obj.d);

#012 }

#013 }
```

图 9：动态变更 field 内容

Java 源码改动办法

先前我曾提到，原本想借由“改动 Java 标准库源码”来测知 Class object 的生成，但由于其 ctor 原始设计为 private，也就是说不可能透过这个管道生成 Class object（而是由 class loader 负责生成），因此“在 ctor 中打印出某种信息”的企图也就失去了意义。

这里我要谈点题外话：如何修改 Java 标准库源码并让它反应到我们的应用程序来。假设我想修改 java.lang.Class，让它在某些情况下打印某种 信息。首先必须找出标准源码！当你下载 JDK 套件并安装妥当，你会发现 jdk150srcjavalang 目录（见图 10）之中有 Class.java，这就是我们此次行动的标准源码。备份后加以修改，编译获得 Class.class。接下来准备将 .class 搬移到 jdk150jrelibendorsed（见图 10）。

这是一个十分特别的目录，class loader 将优先从该处读取内含 classes 的.jar 文件——成功的条件是.jar 内的 classes 压缩路径必须和 Java 标准库的路径完全相 同。为此，我们可以将刚才做出的 Class.class 先搬到一个为此目的而刻意做出来的 javalang 目录中，压缩为 foo.zip（任意命名，唯需 夹带路径 javalang），再将这个 foo.zip 搬到 jdk150jrelibendorsed 并改名为 foo.jar。此后你的应用程序便会优先用 上这里的 java.lang.Class。整个过程可写成一个批处理文件（batch file），如图 11，在 DOS Box 中使用。

图 10：JDK1.5 安装后的目录组织。其中的 endorsed 是我新建。

```
del e:javalang*.class //清理干净

del c:jdk150jrelibendorsedfoo.jar //清理干净

c:

cd c:jdk150srcjavalang

javac -Xlint:unchecked Class.java //编译源码

javac -Xlint:unchecked ClassLoader.java //编译另一个源码（如有必要）

move *.class e:javalang //搬移至刻意制造的目录中

e:

cd e:javalang //以下压缩至适当目录

pkzipc -add -path=root c:jdk150jrelibendorsedfoo.jar *.class

cd e:          est //进入测试目录

javac -Xlint:unchecked Test.java //编译测试程序

java Test //执行测试程序
```

图 11：一个可在 DOS Box 中使用的批处理文件（batch file），用以自动化 java.lang.Class 的修改动作。Pkzipc(.exe)是个命令列压缩工具，add 和 path 都是其命令。

更多信息

以下是与本文主题相关的更多讨论。这些信息可以弥补因文章篇幅限制而带来的不足

- "Take an in-depth look at the Java Reflection API -- Learn about the new Java 1.1 tools forfinding out information about classes", by Chuck McManis。此篇文章所附程序代码是本文示例程序的主要依据（本文示例程序示范了更多 Reflection APIs，并采用 JDK1.5 新式的 for-loop 写法）。
- "Take a look inside Java classes -- Learn to deduce properties of a Java class from

inside aJava program", by Chuck McManis。

l "The basics of Java class loaders -- The fundamentals of this key component of the Javaarchitecture", by Chuck McManis。

l 《The Java Tutorial Continued》, Sun microsystems. Lesson58-61, "Reflection".

注 1 用过诸如 MFC 这类所谓 Application Framework 的程序员也许知道，MFC 有所谓的 dynamic creation。但它并不等同于 Java 的动态加载或动态辨识；所有能够在 MFC 程序中起作用的 classes，都必须先在编译期被编译器“看见”。

注 2 如果操作对象是 Object，Class.getSuperClass()会返回 null。

【发表评论】【加入收藏】【告诉好友】【打印此文】【关闭窗口】
评论人 评论内容 评论时间 打分
【suifeng】 查看所有导入的类，上面介绍的方法，不能看到 在方法内部的类。
如：
```
public void test(){
    MyClass a = new MyClass();
}
```
对于 MyClass 类就不能看到。
2007-9-25 11:34:30 3 分
【jack】 写得很不错。但是我不明白，如果能够获得该类的 class 为什么还要用反射的 function 呢？直接通过.newInstance 不就可以获得对象实例 吗？获取对象实例不就可以操作任何的成员和变量吗？请高手拿出一些确切可以反映 reflect 好处的实用代码。
还有一个问题。就是 public static Class forName(String name, boolean initialize,ClassLoader loader)。loader 可以从本类通过 getClassLoader 获取，也可以通过 classLoader = Thread.currentThread().getContextClassLoader();获得。有什么区别？ 2007-9-21 11:03:56 3 分
【laurent】 学习了，转载到我的 blog 上
谢谢了~~~ 2007-5-24 14:58:41 5 分
【小康】 例子代码有问题的。
图 9 中，
#010 f.set(obj, 12.34);
是无法通过编译的。。。
应该改为
f.set(obj,Double.valueOf("12.34"));
修改后执行成功。 2006-8-8 16:40:32 1 分

java 反射机制 2

JAVA 反射机制

JAVA 反射机制是在运行状态中，对于任意一个类，都能够知道这个类的所有属性和方法；对于任意一个对象，都能够调用它的任意一个方法；这种动态获取的信息以及动态调用对象的方法的功能称为 java 语言的反射机制。
Java 反射机制主要提供了以下功能： 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。
1. 得到某个对象的属性

```
1 public Object getProperty(Object owner, String fieldName) throws Exception {
2        Class ownerClass = owner.getClass();
3
4        Field field = ownerClass.getField(fieldName);
5
6        Object property = field.get(owner);
7
8        return property;
9 }
```

Class ownerClass = owner.getClass()：得到该对象的 Class。

Field field = ownerClass.getField(fieldName)：通过 Class 得到类声明的属性。

Object property = field.get(owner)：通过对象得到该属性的实例，如果这个属性是非公有的，这里会报 IllegalAccessException。

2. 得到某个类的静态属性

```
1 public Object getStaticProperty(String className, String fieldName)
2                 throws Exception {
3        Class ownerClass = Class.forName(className);
4
5        Field field = ownerClass.getField(fieldName);
6
7        Object property = field.get(ownerClass);
8
9        return property;
10 }
```

Class ownerClass = Class.forName(className) ： 首先得到这个类的 Class。

Field field = ownerClass.getField(fieldName)：和上面一样，通过 Class 得到类声明的属性。

Object property = field.get(ownerClass) ：这里和上面有些不同，因为该属性是静态的，所以直接从类的 Class 里取。

3. 执行某对象的方法

```
1 public Object invokeMethod(Object owner, String methodName, Object[] args) throws Exception {
2
3       Class ownerClass = owner.getClass();
4
5       Class[] argsClass = new Class[args.length];
6
7       for (int i = 0, j = args.length; i < j; i++) {
8             argsClass  = args.getClass();
9       }
10
11       Method method = ownerClass.getMethod(methodName, argsClass);
12
13       return method.invoke(owner, args);
14 }
```

Class owner_class = owner.getClass() ： 首先还是必须得到这个对象的 Class。

5～9行：配置参数的 Class 数组，作为寻找 Method 的条件。

Method method = ownerClass.getMethod(methodName, argsClass)：通过 Method 名和参数的 Class 数组得到要执行的 Method。

method.invoke(owner, args)：执行该 Method，invoke 方法的参数是执行这个方法的对象，和参数数组。返回值是 Object，也既是该方法的返回值。

4. 执行某个类的静态方法

```
1 public Object invokeStaticMethod(String className, String methodName,
2                   Object[] args) throws Exception {
3       Class ownerClass = Class.forName(className);
4
5       Class[] argsClass = new Class[args.length];
6
7       for (int i = 0, j = args.length; i < j; i++) {
```

```
8               argsClass  = args.getClass();
9       }
10
11       Method method = ownerClass.getMethod(methodName, argsClass);
12
13       return method.invoke(null, args);
14 }
```

基本的原理和实例3相同，不同点是最后一行，invoke 的一个参数是 null，因为这是静态方法，不需要借助实例运行。

5. 新建实例

```
1
2 public Object newInstance(String className, Object[] args) throws Exception {
3       Class newoneClass = Class.forName(className);
4
5       Class[] argsClass = new Class[args.length];
6
7       for (int i = 0, j = args.length; i < j; i++) {
8               argsClass  = args.getClass();
9       }
10
11       Constructor cons = newoneClass.getConstructor(argsClass);
12
13       return cons.newInstance(args);
14
15 }
```

这里说的方法是执行带参数的构造函数来新建实例的方法。如果不需要参数，可以直接使用newoneClass.newInstance()来实现。

Class newoneClass = Class.forName(className)：第一步，得到要构造的实例的 Class。

第5～第9行：得到参数的 Class 数组。

Constructor cons = newoneClass.getConstructor(argsClass)：得到构造子。

cons.newInstance(args)：新建实例。

6. 判断是否为某个类的实例

```
1 public boolean isInstance(Object obj, Class cls) {
2       return cls.isInstance(obj);
3 }
```

7. 得到数组中的某个元素

```
1 public Object getByArray(Object array, int index) {
2         return Array.get(array,index);
3 }
```

反射数组

在 Java 语言中数组是对象，象其它所有的对象一样，它有一些类。如果你有一个数组，你可以和其它任何对象一样使用标准的 getClass 方法来获得这个 数组的类，但是你获得的这个类与其它的对象类型相比，不同之处在它没有一个现存的工作实例。即使你有了一个数组类之后，你也不能够直接用它来做任何事情， 因为通过反射为普通的类所提供的构造器访问不能为数组工作，并且数组没有任何可访问的属性字段，只有基本的为数组对象定义的 java.lang.Object 类型的方法。

数组特殊处理要使用 java.lang.reflect.Array 类提供的一个静态方法的集合，这个类中的方法可以让你创建新的数组，获得一个数组对象的长度，以及读写一个数组对象的索引值。

下面的代码显示了有效调整一个现存数组的尺寸的方法。它使用反射来创建一个相同类型的新数组，然后在返回这个新数组之前把原数组中的所有的数据复制到新的数组中。

```
public Object growArray(Object array, int size) {
    Class type = array.getClass().getComponentType();
    Object grown = Array.newInstance(type, size);
    System.arraycopy(array, 0, grown, 0,
        Math.min(Array.getLength(array), size));
    return grown;
}
```

安全与反射

在处理反射的时候，安全是一个复杂的问题。反射正常被框架类型的代码使用，并因为这样，你可能会经常要求框架不关心普通的访问限制来完全访问你的代码。然而，自由的访问可能会在其它的一些实例中产生一些风险，例如在代码在一个不被信任的代码共享环境中被执行的时候。

因为这些冲突的需要，Java 语言定义了一个多级方法来处理反射安全。基本的模式是在反射请求源码访问的时候强制使用如下相同的约束限制：

访问这个类中来自任何地方的 public 组件；
不访问这个类本身外部的 private 组件；
限制访问 protected 和 package(默认访问)组件。

围绕这些限制有一个简单的方法，我在前面的例子中所使用的所有构造器、属性字段、以及类的方法都扩展于一个共同的基类??? java.lang.reflect.AccessibleObject 类。这个类定义了一个 setAccessible 方法，这个方法可以让你打开或关 闭这些对类的实例的访问检查。如果安全管理器被设置为关闭访问检查，那么就允许你访问，否则不允许，安全管理器会抛出一个异常。

下面是一个使用反向来演示这种行为的 TwoString 类的实例。

```
public class ReflectSecurity {
    public static void main(String[] args) {
        try {
            TwoString ts = new TwoString("a", "b");
            Field field = clas.getDeclaredField("m_s1");
//          field.setAccessible(true);
            System.out.println("Retrieved value is " +
                field.get(inst));
        } catch (Exception ex) {
            ex.printStackTrace(System.out);
        }
    }
}
```

如果你编译这段代码并且直接使用不带任何参数的命令行命令来运行这个程序，它会抛出一个关于 field.get(inst) 调用的 IllegalAccessException 异常，如果你去掉上面代码中 field.setAccessible(true)行的注释，然后编译并重新 运行代码，它就会成功执行。最后，如果你在命令行给 JVM 添加一个 Djava.security.manager 参数，使得安全管理器可用，那么它又会失 败，除非你为 ReflectSecurity 类定义安全许可。

反射性能

反射是一个强大的工具，但是也会带一些缺点。主要缺点之一就是对性能的影响。使用反射是基本的解释性操作，你告诉 JVM 你要做什么，它就会为你做什么。这 种操作类型总是比直接做同样的操作要慢。为了演示使用反射所要付出的性能代价，我为这篇文章准备了一套基准程序（可以从资源中下载）。

下面列出一段来自于属性字段的访问性能测试的摘要，它包括基本的测试方法。每个方法测试一种访问属性字段的形式，accessSame 方法和本对象的成员 字段一起工作，accessReference 方法直接使用另外的对象属性字段来存取，accessReflection 通过反射使用另一个对象的属性字 段来存取，每个方法都使用相同的计算???在循环中简单的加/乘运算。

```
public int accessSame(int loops) {
    m_value = 0;
    for (int index = 0; index < loops; index++) {
        m_value = (m_value + ADDITIVE_VALUE) *
            MULTIPLIER_VALUE;
    }
    return m_value;
}

public int accessReference(int loops) {
    TimingClass timing = new TimingClass();
    for (int index = 0; index < loops; index++) {
        timing.m_value = (timing.m_value + ADDITIVE_VALUE) *
            MULTIPLIER_VALUE;
    }
    return timing.m_value;
}

public int accessReflection(int loops) throws Exception {
    TimingClass timing = new TimingClass();
    try {
        Field field = TimingClass.class.
            getDeclaredField("m_value");
        for (int index = 0; index < loops; index++) {
            int value = (field.getInt(timing) +
                ADDITIVE_VALUE) * MULTIPLIER_VALUE;
            field.setInt(timing, value);
        }
        return timing.m_value;
    } catch (Exception ex) {
        System.out.println("Error using reflection");
        throw ex;
    }
}
```

测试程序在一个大循环中反复的调用每个方法，在调用结束后计算平均时间。每个方法的第一次调用不包括在平均值中，因些初始化时间不是影响结果的因素。为这 篇文章所做的测试运行，我为每个调用使用了10000000的循环计数，代码运行在1GHz PIII 系统上。并且分别使用了三个不同的 Linux JVM，对于每个 JVM 都使用了默认设置，测试结果如下图所示：

上面的图表的刻度可以显示整个测试范围，但是那样的话就会减少差别的显示效果。这个图表中的前两个是用 SUN 的 JVM 的进行测试的结果图，使用反射的执行 时间比使用直接访

问的时间要超过1000多倍。最后一个图是用 IBM 的 JVM 所做的测试，通过比较要 SUN 的 JVM 执行效率要高一些，但是使用反射的方法 依然要比其它方法超出700多倍。虽然 IBM 的 JVM 要比 SUN 的 JVM 几乎要快两倍，但是在使用反射之外的两种方法之间，对于任何的 JVM 在执行效率上 没有太大的差别。最大的可能是，这种差别反映了通过 Sun Hot Spot JVMs 在简化基准方面所做的专门优化很少。

除了属性字段访问时间的测试以外，我对方法做了同样的测试。对于方法的调用，我偿试了与属性字段访问测试一样的三种方式，用额外使用了没有参数的方法的变量与传递并返回一个值的方法调用相对比。下面的代码显示了使用传递并返回值的调用方式进行测试的三种方法。

```
public int callDirectArgs(int loops) {
    int value = 0;
    for (int index = 0; index < loops; index++) {
        value = step(value);
    }
    return value;
}

public int callReferenceArgs(int loops) {
    TimingClass timing = new TimingClass();
    int value = 0;
    for (int index = 0; index < loops; index++) {
        value = timing.step(value);
    }
    return value;
}

public int callReflectArgs(int loops) throws Exception {
    TimingClass timing = new TimingClass();
    try {
        Method method = TimingClass.class.getMethod
            ("step", new Class [] { int.class });
        Object[] args = new Object[1];
        Object value = new Integer(0);
        for (int index = 0; index < loops; index++) {
            args[0] = value;
            value = method.invoke(timing, args);
        }
        return ((Integer)value).intValue();
    } catch (Exception ex) {
        System.out.println("Error using reflection");
        throw ex;
    }
}
```

下图显示我使用这些方法的测试结果，这里再一次显示了反射要比其它的直接访问要慢很多。虽然对于无参数的案例，执行效率从 SUN1.3.1JVM 的慢几百 倍到 IBM 的 JVM 慢不到30倍，与属性字段访问案例相比，差别不是很大，这种情况的部分原因是因为 java.lang.Integer 的包装器需要传递 和返回 int 类型的值。因为 Intergers 是不变的，因此就需要为每个方法的返回生成一个新值，这就增加了相当大的系统开销。

反射的性能是 SUN 在开发1.4JVM 时重点关注的一个领域，从上图可以看到改善的结果。Sun1.4.1JVM 对于这种类型的操作比1.3.1版有了很 大的提高，要我的测试中要快大约7倍。IBM 的1.4.0JVM 对于这种测试提供了更好的性能，它的运行效率要比 Sun1.4.1JVM 快两到三倍。

我还为使用反射创建对象编写了一个类似的效率测试程序。虽然这个例子与属性字段和方法调用相比差别不是很大，但是在 Sun1.3.1JVM 上调用 newInstance（）方法创建一个简单的 java.lang.Object 大约比直接使用 new Object()方法长12倍的时间，在 IBM1.4.0JVM 上大约要长4倍的时间，在 Sun1.4.1JVM 上大约要长2倍的时间。对于任何用于测试 的 JVM，使用 Array.newInstance(Type,size)方法创建一个数组所需要的时间比使用 new tye[size]所花费的时间大约要长两倍，随着数组民尺寸的增长，这两种方法的差别的将随之减少。

反射概要总结

Java 语言的反射机制提供了一种非常通用的动态连接程序组件的方法。它允许你的程序创建和维护任何类的对象（服从安全限制），而不需要提前对目标类进行硬编码。 这些特征使得反射在创建与对象一同工作的类库中的通用方法方面非常有用。例如，反射经常被用于那些数据库，XML、或者其它的外部的持久化对象的框架中。

反射还有两个缺点，一个是性能问题。在使用属性字段和方法访问的时候，反射要比直接的代码访问要慢很多。至于对影响的程度，依赖于在程序中怎样使用反射。 如果它被用作一个相关的很少发生的程序操作中，那么就不必关心降低的性能，即使在我的测试中所展示的最耗时的反射操作的图形中也只是几微秒的时间。如果要 在执行应用程序的核心逻辑中使用反射，性能问题才成为一个要严肃对象的问题。

对于很多应用中的存在的缺点是使用反射可以使你的实际的代码内部逻辑变得模糊不清。程序员都希望在源代码中看到一个程序的逻辑以及象绕过源代码的反射所可 能产生的维护问题这样的一些技术。反射代码也比相应的直接代码要复杂一些，就像在性能比较的代码实例看到那样。处理这些问题的最好方法是尽可能少使用反 射，只有在一些增加灵活性的地方来使用它。

在下一篇文章中，我将给出一个更加详细的如何使用反射的例子。这个例子提供了一个用于处理传递给一个 Java 应用程序的命令行参数的 API。在避免弱点的 同时，它也显示了反射

的强大的功能，反射能够使用的你的命令处理变得的简单吗？你可以在 Java 动态程序设计的第三部分中找到答案。

2.开始使用 Reflection

用于 reflection 的类，如 Method，可以在 java.lang.relfect 包中找到。使用这些类的时候必须要遵循三个步骤：第一步是获得你想操作的类的 java.lang.Class 对象。在运行中的 Java 程序中，用 java.lang.Class 类来描述类和接口等。

下面就是获得一个 Class 对象的方法之一：

```
Class c = Class.forName("java.lang.String");
```

这条语句得到一个 String 类的类对象。还有另一种方法，如下面的语句：

```
Class c = int.class;
```

或者

```
Class c = Integer.TYPE;
```

它们可获得基本类型的类信息。其中后一种方法中访问的是基本类型的封装类 (如 Integer) 中预先定义好的 TYPE 字段。

第二步是调用诸如 getDeclaredMethods 的方法，以取得该类中定义的所有方法的列表。

一旦取得这个信息，就可以进行第三步了——使用 reflection API 来操作这些信息，如下面这段代码：

```
Class c = Class.forName("java.lang.String");
Method m[] = c.getDeclaredMethods();
System.out.println(m[0].toString());
```

它将以文本方式打印出 String 中定义的第一个方法的原型。

在下面的例子中，这三个步骤将为使用 reflection 处理特殊应用程序提供例证。

模拟 instanceof 操作符

得到类信息之后，通常下一个步骤就是解决关于 Class 对象的一些基本的问题。例如，Class.isInstance 方法可以用于模拟 instanceof 操作符：

```
class A {
}
public class instance1 {
public static void main(String args[]) {
try {
Class cls = Class.forName("A");
boolean b1 = cls.isInstance(new Integer(37));
System.out.println(b1);
boolean b2 = cls.isInstance(new A());
System.out.println(b2);
} catch (Throwable e) {
System.err.println(e);
}
}
}
```

在这个例子中创建了一个 A 类的 Class 对象，然后检查一些对象是否是 A 的实例。Integer(37) 不是，但 new A() 是。

3.找出类的方法

找出一个类中定义了些什么方法，这是一个非常有价值也非常基础的 reflection 用法。下面的代码就实现了这一用法：

```
import java.lang.reflect.*;
public class method1 {
private int f1(Object p, int x) throws NullPointerException {
if (p == null)
throw new NullPointerException();
return x;
}
public static void main(String args[]) {
try {
Class cls = Class.forName("method1");
Method methlist[] = cls.getDeclaredMethods();
for (int i = 0; i < methlist.length; i++) {
Method m = methlist[i];
System.out.println("name = " + m.getName());
System.out.println("decl class = " + m.getDeclaringClass());
Class pvec[] = m.getParameterTypes();
for (int j = 0; j < pvec.length; j++)
System.out.println("param #" + j + " " + pvec[j]);
Class evec[] = m.getExceptionTypes();
for (int j = 0; j < evec.length; j++)
System.out.println("exc #" + j + " " + evec[j]);
System.out.println("return type = " + m.getReturnType());
System.out.println("-----");
}
} catch (Throwable e) {
System.err.println(e);
}
}
}
```

这个程序首先取得 method1 类的描述，然后调用 getDeclaredMethods 来获取一系列的 Method 对象，它们分别描述了定义在类中的每一个方法，包括 public 方法、protected 方法、package 方法和 private 方法等。如果你在程序中使用 getMethods 来代替 getDeclaredMethods，你还能获得继承来的各个方法的信息。

取得了 Method 对象列表之后，要显示这些方法的参数类型、异常类型和返回值类型等就不难了。这些类型是基本类型还是类类型，都可以由描述类的对象按顺序给出。

输出的结果如下：

```
name = f1
decl class = class method1
```

```
param #0 class java.lang.Object
param #1 int
exc #0 class java.lang.NullPointerException
return type = int
-----
name = main
decl class = class method1
param #0 class [Ljava.lang.String;
return type = void
-----
```

4.获取构造器信息

5.获取类的字段(域)

找出一个类中定义了哪些数据字段也是可能的，下面的代码就在干这个事情：

```
import java.lang.reflect.*;
public class field1 {
private double d;
public static final int i = 37;
String s = "testing";
public static void main(String args[]) {
try {
Class cls = Class.forName("field1");
Field fieldlist[] = cls.getDeclaredFields();
for (int i = 0; i < fieldlist.length; i++) {
Field fld = fieldlist;
System.out.println("name = " + fld.getName());
System.out.println("decl class = " + fld.getDeclaringClass());
System.out.println("type = " + fld.getType());
int mod = fld.getModifiers();
System.out.println("modifiers = " + Modifier.toString(mod));
System.out.println("-----");
}
} catch (Throwable e) {
System.err.println(e);
}
}
}
```

这个例子和前面那个例子非常相似。例中使用了一个新东西 Modifier，它也是一个 reflection 类，用来描述字段成员的修饰语，如“private int”。这些修饰语自身由整数描述，而且使用 Modifier.toString 来返回以“官方”顺序排列的字符串描述 (如“static”在“final”之前)。这个程序的输出是：

```
name = d
decl class = class field1
type = double
modifiers = private
-----
name = i
decl class = class field1
type = int
modifiers = public static final
-----
name = s
decl class = class field1
type = class java.lang.String
modifiers =
-----
```

和获取方法的情况一下，获取字段的时候也可以只取得在当前类中申明了的字段信息(getDeclaredFields)，或者也可以取得父类中定义的字段 (getFields) 。

6.根据方法的名称来执行方法

文本到这里，所举的例子无一例外都与如何获取类的信息有关。我们也可以用 reflection 来做一些其它的事情，比如执行一个指定了名称的方法。下面的示例演示了这一操作：

```
import java.lang.reflect.*;
public class method2 {
public int add(int a, int b) {
return a + b;
}
public static void main(String args[]) {
try {
Class cls = Class.forName("method2");
Class partypes[] = new Class[2];
partypes[0] = Integer.TYPE;
partypes[1] = Integer.TYPE;
Method meth = cls.getMethod("add", partypes);
method2 methobj = new method2();
Object arglist[] = new Object[2];
arglist[0] = new Integer(37);
arglist[1] = new Integer(47);
Object retobj = meth.invoke(methobj, arglist);
Integer retval = (Integer) retobj;
System.out.println(retval.intvalue());
} catch (Throwable e) {
```

```
System.err.println(e);
}
}
}
```

假如一个程序在执行的某处的时候才知道需要执行某个方法，这个方法的名称是在程序的运行过程中指定的 (例如，JavaBean 开发环境中就会做这样的事)，那么上面的程序演示了如何做到。
上例中，getMethod 用于查找一个具有两个整型参数且名为 add 的方法。找到该方法并创建了相应的 Method 对象之后，在正确的对象实例中执行它。执行该方法的时候，需要提供一个参数列表，这在上例中是分别包装了整数 37 和 47 的两个 Integer 对象。执行方法的返回的同样是一个 Integer 对象，它封装了返回值 84。

7.创建新的对象
对于构造器，则不能像执行方法那样进行，因为执行一个构造器就意味着创建了一个新的对象 (准确的说，创建一个对象的过程包括分配内存和构造对象)。所以，与上例最相似的例子如下：

```
import java.lang.reflect.*;
public class constructor2 {
public constructor2() {
}
public constructor2(int a, int b) {
System.out.println("a = " + a + " b = " + b);
}
public static void main(String args[]) {
try {
Class cls = Class.forName("constructor2");
Class partypes[] = new Class[2];
partypes[0] = Integer.TYPE;
partypes[1] = Integer.TYPE;
Constructor ct = cls.getConstructor(partypes);
Object arglist[] = new Object[2];
arglist[0] = new Integer(37);
arglist[1] = new Integer(47);
Object retobj = ct.newInstance(arglist);
} catch (Throwable e) {
System.err.println(e);
}
}
}
```

根据指定的参数类型找到相应的构造函数并执行它，以创建一个新的对象实例。使用这种方法可以在程序运行时动态地创建对象，而不是在编译的时候创建对象，这一点非常有价值。

8.改变字段(域)的值

reflection 的还有一个用处就是改变对象数据字段的值。reflection 可以从正在运行的程序中根据名称找到对象的字段并改变它，下面的例子可以说明这一点：

```
import java.lang.reflect.*;
public class field2 {
public double d;
public static void main(String args[]) {
try {
Class cls = Class.forName("field2");
Field fld = cls.getField("d");
field2 f2obj = new field2();
System.out.println("d = " + f2obj.d);
fld.setDouble(f2obj, 12.34);
System.out.println("d = " + f2obj.d);
} catch (Throwable e) {
System.err.println(e);
}
}
}
```

这个例子中，字段 d 的值被变为了 12.34。

9.使用数组

本文介绍的 reflection 的最后一种用法是创建的操作数组。数组在 Java 语言中是一种特殊的类类型，一个数组的引用可以赋给 Object 引用。观察下面的例子看看数组是怎么工作的：

```
import java.lang.reflect.*;
public class array1 {
public static void main(String args[]) {
try {
Class cls = Class.forName("java.lang.String");
Object arr = Array.newInstance(cls, 10);
Array.set(arr, 5, "this is a test");
String s = (String) Array.get(arr, 5);
System.out.println(s);
} catch (Throwable e) {
System.err.println(e);
}
}
}
```

例中创建了 10 个单位长度的 String 数组，为第 5 个位置的字符串赋了值，最后将这个字符串从数组中取得并打印了出来。

下面这段代码提供了一个更复杂的例子：

```
import java.lang.reflect.*;
public class array2 {
public static void main(String args[]) {
```

```
int dims[] = new int[]{5, 10, 15};
Object arr = Array.newInstance(Integer.TYPE, dims);
Object arrobj = Array.get(arr, 3);
Class cls = arrobj.getClass().getComponentType();
System.out.println(cls);
arrobj = Array.get(arrobj, 5);
Array.setInt(arrobj, 10, 37);
int arrcast[][][] = (int[][][]) arr;
System.out.println(arrcast[3][5][10]);
}
}
```

例中创建了一个 5 x 10 x 15 的整型数组，并为处于 [3][5][10] 的元素赋了值为 37。注意，多维数组实际上就是数组的数组，例如，第一个 Array.get 之后，arrobj 是一个 10 x 15 的数组。进而取得其中的一个元素，即长度为 15 的数组，并使用 Array.setInt 为它的第 10 个元素赋值。

注意创建数组时的类型是动态的，在编译时并不知道其类型。

JAVA 反射机制(转)

该篇文章介绍了一些 Java 二进制类格式的相关信息。这个月我将阐述使用 Java 反射 API 来在运行时接入和使用一些相同信息的基础。为了使已经熟知反射基础的开发人员关注本文，我将在文章中包括反射性能如何与直接接入相比较。

使用反射不同于常规的 Java 编程，其中它与元数据--描述其它数据的数据协作。Java 语言反射接入的特殊类型的原数据是 JVM 中类和对象的描述。反射使您能够运行时接入广泛的类信息。它甚至使您能够读写字段,调用运行时选择的类的方法。

反射是一种强大的工具。它使您能够创建灵活的代码，这些代码可以在运行时装配，无需在组件之间进行源代表链接。但反射的某些方面存在一些疑问。在本文中， 我将深入讨论为什么您可能不希望在程序中使用反射，以及您应该这样做的理由。在了解了权衡性分析之后，您可以自行决定是否利大于弊。

初学者的类

使用反射的启点总是 java.lang.Class 实例。如果您希望与预先定义的类协作，那么 Java 语言提供一种直接获得 Class 实例的简便快捷方式：

```
Class clas = MyClass.class;
```

当您使用这一项技术时，装入类涉及的所有工作在幕后进行。但是，如果您需要在运行时从某些外部源读取类名，这种方法并不适合。实际上，您需要使用一个类装入器来查找类信息。以下介绍一种方法：

```
// "name" is the class name to load

Class clas = null;

try {

   clas = Class.forName(name);

} catch (ClassNotFoundException ex) {

   // handle exception case
```

```
}

// use the loaded class
```

如果已经装入了类，您将得到现有的 Class 信息。如果类未被装入，类装入器将现在装入并返回新创建的类实例。

类的反射

Class 对象为您提供接入类元数据的反射的所有基本 hook。这类元数据包括关于类自身的信息，如包和类的父类，以及该类实施的接口。它还包括该类定义的构造函 数、字段和方法的详细信息。这些最后的项目都是编程中最经常使用的项目，因此我将在本小节的稍后部分给出一些与它们协作的实例。

对于以下三类组件中的任何一类来说 -- 构造函数、字段和方法 -- java.lang.Class 提供四种独立的反射调用，以不同的方式来获得信息。调用都遵循一种标准格式。以下是用于查找构造函数的一组反射调用：

- Constructor getConstructor(Class[] params) -- 获得使用特殊的参数类型的公共构造函数，
- Constructor[] getConstructors() -- 获得类的所有公共构造函数
- Constructor getDeclaredConstructor(Class[] params) -- 获得使用特定参数类型的构造函数(与接入级别无关)
- Constructor[] getDeclaredConstructors() -- 获得类的所有构造函数(与接入级别无关)

每类这些调用都返回一个或多个 java.lang.reflect.Constructor 函数。这种 Constructor 类定义 newInstance 方法，它采用一组对象作为其唯一的参数，然后返回新创建的原始类实例。该组对象是用于构造函数调用的参数值。作为解释这一工作流程的实例，假设您有一个 TwoString 类和一个使用一对 String s 的构造函数，如清单1所示：

清单1:从一对字符串创建的类

```
public class TwoString {

    private String m_s1, m_s2;

    public TwoString(String s1, String s2) {

        m_s1 = s1;
```

```
        m_s2 = s2;
    }
}
```

清单2中的代码获得构造函数并使用它来创建使用 String s "a" 和 "b" 的 TwoString 类的一个实例:

清单2：构造函数的反射调用

```
Class[] types = new Class[] { String.class, String.class };
Constructor cons = TwoString.class.getConstructor(types);
Object[] args = new Object[] { "a", "b" };
TwoString ts = cons.newInstance(args);
```

清单2中的代码忽略了不同反射方法抛出的多种可能选中的例外类型。例外在 Javadoc API 描述中详细记录，因此为了简明起见，我将在所有程序实例中忽略它们。

尽管我在讨论构造函数主题，Java 编程语言还定义了一种您可以用来使用无参数（或缺省）构造函数创建类的一个实例的特殊快捷方式。这种快捷方式嵌入到 Class 定义中，如下：

Object newInstance() -- 使用缺省函数创建新的实例

即使这种方法只允许您使用一种特殊的构造函数，如果这正是您需要的，那么它将提供一种非常方便的快捷方式。当与 JavaBeans 协作时这项技术尤其有用，JavaBeans 需要定义公共、无参数构造函数。

通过反射增加字段

获得字段信息的 Class 反射调用不同于那些用于接入构造函数的调用，在参数类型数组中使用了字段名：

- Field getField(String name) -- 获得命名的公共字段
- Field[] getFields() -- 获得类的所有公共字段
- Field getDeclaredField(String name) -- 获得类声明的命名的字段
- Field[] getDeclaredFields() -- 获得类声明的所有字段

尽管与构造函数调用类似，在字段方面仍存在一个重要的区别：前两个变量返回可以通过类接入的公共字段的信息 -- 即使它们来自于祖先类。后两个变量返回类直接声明的字段的信息 -- 与字段的接入类型无关。

调用返回的 java.lang.reflect.Field 实例定义所有主类型的 getXXX 和 setXXX 方法，以及与对象引用协作的通用 get 和 set 方法。您可以根据实际的字段类型自行选择一种适当的方法，而 getXXX 方法将自动处理扩展转换(如使用 getInt 方法来检索一个字节值)。

清单3显示使用字段反射方法的一个实例，以方法的格式根据名称增加对象的 int 字段 ：

清单3：通过反射增加一个字段

```
public int incrementField(String name, Object obj) throws... {

    Field field = obj.getClass().getDeclaredField(name);

    int value = field.getInt(obj) + 1;

    field.setInt(obj, value);

    return value;

}
```

这种方法开始展示了反射带来的某些灵活性。与特定的类协作不同， incrementField 使用传入的对象的 getClass 方法来查找类信息，然后直接在该类中查找命名的字段。

通过反射增加方法

获得方法信息的 Class 反射调用与用于构造函数和字段的调用非常类似：

- Method getMethod(String name, Class[] params) -- 使用特定的参数类型，获得命名的公共方法
- Method[] getMethods() -- 获得类的所有公共方法

- Method getDeclaredMethod(String name, Class[] params) -- 使用特写的参数类型，获得类声明的命名的方法
- Method[] getDeclaredMethods() -- 获得类声明的所有方法

与字段调用一样，前两个变量返回可以通过类接入的公共方法的信息 -- 即使它们来自于祖先类。后两个变量返回类声明的方法的信息，与方法的接入类型无关。

调用返回的 java.lang.reflect.Method 实例定义一种 invoke 方法，您可以用来在正在定义的类的一个实例上调用方法。这种 invoke 方法使用两个参数，为调用提供类实例和参数值数组。

清单4进一步阐述字段实例，显示反射正在运行的方法的一个实例。这种方法增加一个定义有 get 和 set 方法的 int JavaBean 属性。例如，如果对象为一个整数 count 值定义了 getCount 和 setCount 方法，您可以在一次调用中向该方法传递“count”作为 name 参数，以增加该值。

清单4：通过反射增加一个 JavaBean 属性

```
public int incrementProperty(String name, Object obj) {

    String prop = Character.toUpperCase(name.charAt(0)) +

        name.substring(1);

    String mname = "get" + prop;

    Class[] types = new Class[] {};

    Method method = obj.getClass().getMethod(mname, types);

    Object result = method.invoke(obj, new Object[0]);

    int value = ((Integer)result).intValue() + 1;

    mname = "set" + prop;

    types = new Class[] { int.class };

    method = obj.getClass().getMethod(mname, types);

    method.invoke(obj, new Object[] { new Integer(value) });

    return value;

}
```

为了遵循 JavaBeans 惯例，我把属性名的首字母改为大写，然后预先考虑 get 来创建读方法名， set 来创建写方法名。JavaBeans 读方法仅返回值，而写方法使用值作为唯一的参数，因此我规定方法的参数类型以进行匹配。最后，该惯例要求方法为公共，因此我使用查找格式，查找类上可调用的公共方法。

这一实例是第一个我使用反射传递主值的实例，因此现在我们来看看它是如何工作的。基本原理很简单：无论什么时候您需要传递主值，只需用相应封装类的一个实例（在 java.lang 包中定义）来替换该类主值。这可以应用于调用和返回。因此，当我在实例中调用 get 方法时，我预计结果为实际 int 属性值的 java.lang.Integer 封装。

反射数组

数组是 Java 编程语言中的对象。与所有对象一样，它们都有类。如果您有一个数组，使用标准 getClass 方法，您可以获得该数组的类，就象任何其它对象一样。但是，不通过现有的实例来获得类不同于其它类型的对象。即使您有一个数组类，您也不能直接对它进行太多的操作 -- 反射为标准类提供的构造函数接入不能用于数组，而且数组没有任何可接入的字段，只有基本的 java.lang.Object 方法定义用于数组对象。

数组的特殊处理使用 java.lang.reflect.Array 类提供的静态方法的集合。该类中的方法使您能够创建新数组，获得数组对象的长度，读和写数组对象的索引值。

清单5显示了一种重新调整现有数组大小的有效方法。它使用反射来创建相同类型的新数组，然后在返回新数组之前，在老数组中复制所有数据。

清单 5：通过反射来扩展一个数组

```
public Object growArray(Object array, int size) {

    Class type = array.getClass().getComponentType();

    Object grown = Array.newInstance(type, size);

    System.arraycopy(array, 0, grown, 0,

        Math.min(Array.getLength(array), size));

    return grown;

}
```

安全性和反射

在处理反射时安全性是一个较复杂的问题。反射经常由框架型代码使用，由于这一点，您可能希望框架能够全面接入您的代码，无需考虑常规的接入限制。但是，在其它情况下，不受控制的接入会带来严重的安全性风险，如当代码在不值得信任的代码共享的环境中运行时。

由于这些互相矛盾的需求，Java 编程语言定义一种多级别方法来处理反射的安全性。基本模式是对反射实施与应用于源代码接入相同的的限制：

- 从任意位置到类公共组件的接入
- 类自身外部无任何到私有组件的接入
- 受保护和打包（缺省接入）组件的有限接入

不过－至少某些时候，围绕这些限制有一种简单的方法。我在前面实例中使用的 Constructor 、 Field 和 Method 类都扩展了一个普通的基本类-- java.lang.reflect.AccessibleObject 类。该类定义一种 setAccessible 方法，使您能够启动或关闭对这些类中其中一个类的实例的接入检测。唯一的问题在于如果使用了安全性管理器，它将检测正在关闭接入检测的代码是否许可了这样做。如果未许可，安全性管理器抛出一个例外。

清单6展示了一个程序，在 清单 1 TwoString 类的一个实例上使用反射来显示安全性正在运行：

清单 6：反射安全性正在运行

```
public class ReflectSecurity {

    public static void main(String[] args) {

        try {

            TwoString ts = new TwoString("a", "b");

            Field field = clas.getDeclaredField("m_s1");

//              field.setAccessible(true);

            System.out.println("Retrieved value is " +

                field.get(inst));

        } catch (Exception ex) {

            ex.printStackTrace(System.out);
```

```
        }
    }
}
```

如果您编译了这一程序，不使用任何特定参数直接从命令行运行，它将在 field.get(inst) 调用中抛出一个 IllegalAccessException 。如果您未注释 field.setAccessible(true) 代码行，那么重新编译并重新运行该代码，它将取得成功。最后，如果您在命令行添加了 JVM 参数 -Djava.security.manager 以实现安全性管理器，它将再次失败，除非您定义了 ReflectSecurity 类的许可权限。

反射性能

反射是一种强大的工具，但也存在一些不足。一个主要的缺点是对性能有影响。使用反射基本上是一种解释操作，您可以告诉 JVM 您希望做什么并且它满足您的要求。这类操作总是慢于只直接执行相同的操作。为了阐述使用反射的性能成本，我为本文准备了一组基准程序(见 参考资料，完整代码链接)。

清单7是字段接入性能测试的一个摘用，包括基本的测试方法。每种方法测试字段接入的一种形式 -- accessSame 与同一对象的成员字段协作， accessOther 使用可直接接入的另一对象的字段， accessReflection 使用可通过反射接入的另一对象的字段。在每种情况下，方法执行相同的计算 -- 循环中简单的加/乘顺序。

清单 7：字段接入性能测试代码

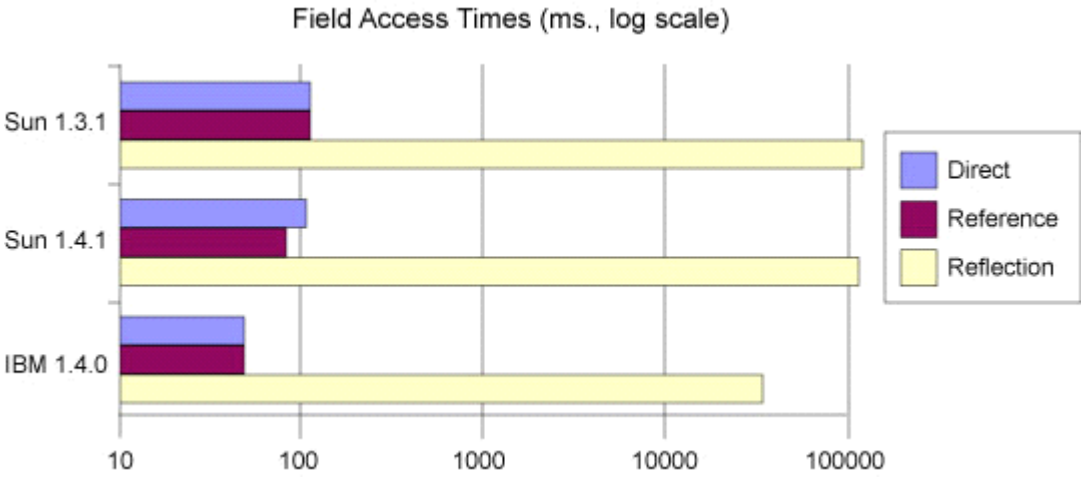
```
public int accessSame(int loops) {
    m_value = 0;
    for (int index = 0; index < loops; index++) {
        m_value = (m_value + ADDITIVE_VALUE) *
            MULTIPLIER_VALUE;
    }
```

```
        return m_value;
    }
    public int accessReference(int loops) {
        TimingClass timing = new TimingClass();
        for (int index = 0; index < loops; index++) {
            timing.m_value = (timing.m_value + ADDITIVE_VALUE) *
                MULTIPLIER_VALUE;
        }
        return timing.m_value;
    }
    public int accessReflection(int loops) throws Exception {
        TimingClass timing = new TimingClass();
        try {
            Field field = TimingClass.class.
                getDeclaredField("m_value");
            for (int index = 0; index < loops; index++) {
                int value = (field.getInt(timing) +
                    ADDITIVE_VALUE) * MULTIPLIER_VALUE;
                field.setInt(timing, value);
            }
            return timing.m_value;
        } catch (Exception ex) {
            System.out.println("Error using reflection");
            throw ex;
```

```
    }

}
```

测试程序重复调用每种方法，使用一个大循环数，从而平均多次调用的时间衡量结果。平均值中不包括每种方法第一次调用的时间，因此初始化时间不是结果中的一个因素。在为本文进行的测试中，每次调用时我使用1000万的循环数，在1GHz PIIIm 系统上运行。三个不同 Linux JVM 的计时结果如图1所示。所有测试使用每个 JVM 的缺省设置。

图 1：字段接入时间



上表的对数尺度可以显示所有时间，但减少了差异看得见的影响。在前两副图中(Sun JVM)，使用反射的执行时间超过使用直接接入的1000倍以上。通过比较，IBM JVM 可能稍好一些，但反射方法仍旧需要比其它方法长700倍以上的时间。任何 JVM 上其它两种方法之间时间方面无任何显著差异，但IBM JVM 几乎比 Sun JVM 快一倍。最有可能的是这种差异反映了 Sun Hot Spot JVM 的专业优化，它在简单基准方面表现得很糟糕。

除了字段接入时间测试之外，我还进行了相同的方法调用时间测试。在方法调用中，我试用了与字段接入相同的三种接入变量，并增加了使用无参数方法变量，而不是在方法调用中传递和返回一个值。清单8显示了用于测试调用传递和返回值形式的三种方法的代码。

清单 8：方法接入性能测试代码

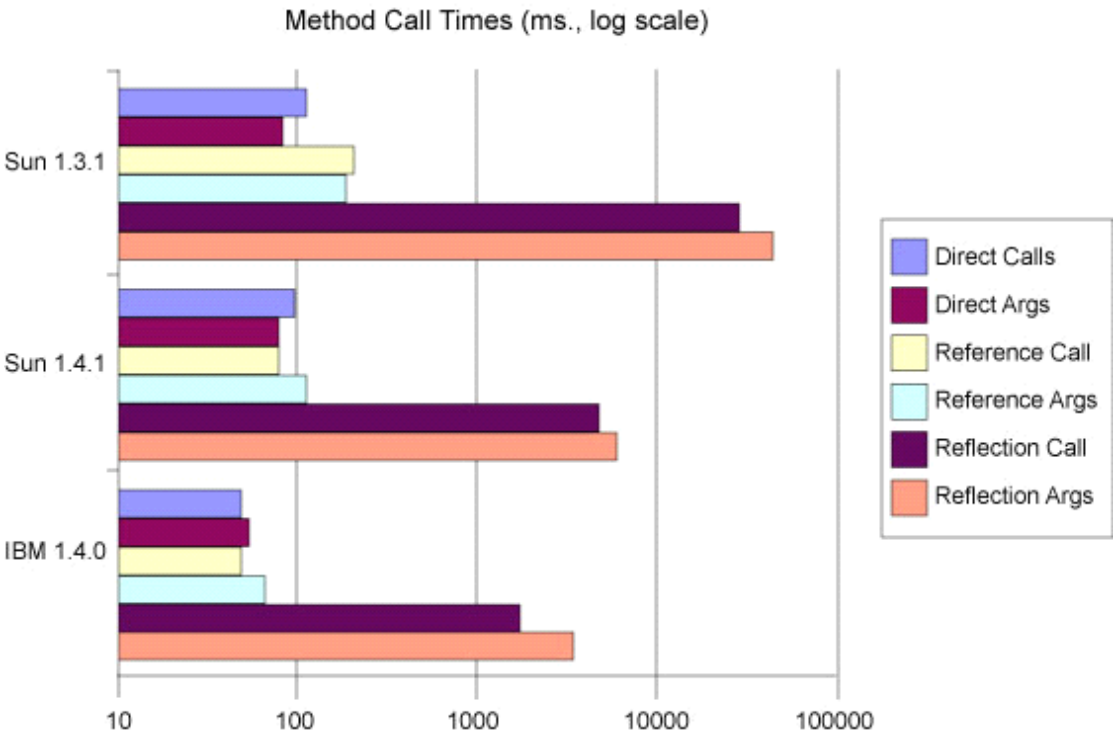
```
public int callDirectArgs(int loops) {

    int value = 0;
```

```
        for (int index = 0; index < loops; index++) {
            value = step(value);
        }
        return value;
    }
    public int callReferenceArgs(int loops) {
        TimingClass timing = new TimingClass();
        int value = 0;
        for (int index = 0; index < loops; index++) {
            value = timing.step(value);
        }
        return value;
    }
    public int callReflectArgs(int loops) throws Exception {
        TimingClass timing = new TimingClass();
        try {
            Method method = TimingClass.class.getMethod
                ("step", new Class [] { int.class });
            Object[] args = new Object[1];
            Object value = new Integer(0);
            for (int index = 0; index < loops; index++) {
                args[0] = value;
                value = method.invoke(timing, args);
            }
```

```
            return ((Integer)value).intValue();
        } catch (Exception ex) {
            System.out.println("Error using reflection");
            throw ex;
        }
}
```

图 2显示了我从方法调用中获得的计时结果。反射远慢于直接接入。差异不象字段接入那么大，但是，在不使用参数的情况下，范围从 Sun 1.3.1 JVM 的数百倍到 IBM JVM 的不到30倍。在所有 JVM 上，使用参数的反射方法调用的测试性能慢于不使用参数的调用。由于传递和返回 int 值需要的 java.lang.Integer 封装，这可能是局部的。由于 Integer s 是不可变的，每种方法返回提出了一种新的需求，它将增加大量的开销。

图 2：方法调用时间



反射性能是 Sun 开发1.4 JVM 时关注的一个方面，它在反射方法调用结果中显示。在这类操作的性能方面， Sun 1.4.1 JVM 显示了比1.3.1版本很大的改进，在我的测试中运行速度大约

是1.3.1版本的开部。在这类简单的测试中，IBM 1.4.0 JVM 再次获得了更好的成绩，但是只比 Sun 1.4.1 JVM 快两到三倍。

我还为创建使用反射的对象编写了类似的计时测试程序，但这种情况下的差异不象字段和方法调用情况下那么显著。使用 newInstance() 调用创建一个简单的 java.lang.Object 实例耗用的时间大约是在 Sun 1.3.1 JVM 上使用 new Object() 的12倍，是在 IBM 1.4.0 JVM 的四倍，只是 Sun 1.4.1 JVM 上的两部。使用 Array.newInstance(type, size) 创建一个数组耗用的时间是任何测试的 JVM 上使用 new type[size] 的两倍，随着数组大小的增加，差异逐步缩小。

结束语

Java 语言反射提供一种动态链接程序组件的多功能方法。它允许程序创建和控制任何类的对象(根据安全性限制)，无需提前硬编码目标类。这些特性使得反射 特别适用于创建以非常普通的方式与对象协作的库。例如，反射经常在持续存储对象为数据库、XML 或其它外部格式的框架中使用。

反射有两个缺点。第一个是性能问题。当用于字段和方法接入时反射要远慢于直接代码。性能问题的程度取决于程序中是如何使用反射的。如果它作为程序运行中相 对很少涉及的部分，缓慢的性能将不会是一个问题。即使测试中最坏情况下的计时图显示的反射操作只耗用几微秒。仅反射在性能关键的应用的核心逻辑中使用时性 能问题才变得至关重要。

许多应用更严重的一个缺点是使用反射会模糊程序内部实际要发生的事情。程序人员希望在源代码中看到程序的逻辑，反射等绕过了源代码的技术会带来维护问题。反射代码比相应的直接代码更复杂，正如性能比较的代码实例中看到的一样。解决这些问题的最佳方案是保守地使用反射-- 仅在它可以真正增加灵活性的地方 -- 记录其在目标类中的使用。

在下一部分，我将提供如何使用反射的更详细实例。这种实例提供一个处理 Java 应用命令行参数的 API，一种您可能发现适用于自己应用的工具。它还基于反射的优势来创建，同时避免其弱点。反射是否能简化您的命令行处理？您可以在 Java 编程的动态性第3部分找到答案。