能够分析类能力的程序称为**反射**。反射机制的功能有：

1. 在运行时分析类的能力。
2. 在运行时检查对象。例如，编写一个适用于所有类的`toString`方法。
3. 实现泛型数组操作代码。
4. 利用`Method`对象。

1 Class类

在程序运行期间，Java运行时系统始终为所有对象维护一个运行时类型标识，这些信息会跟踪每个对象所属的类，虚拟机利用运行时类型信息选择要执行的正确的方法。`Class`类用于保存这些信息。

`Object`类中的`getClass`方法返回一个`Class`类型的实例，`Class`类中的`getName`方法返回类名，这两个方法结合起来可以用来获得类名。如果类在一个包里，包名也会作为类名的一部分。例如：

```
Employee e;
System.out.println(e.getClass().getName());
```

Class中的静态方法`forName`可以获得类名对应的Class对象，它的签名和用法如下：

```
/* 签名 */
static Class forName(String className)
/* 用法 */
String className = "java.util.Random";
Class cl = Class.forName(className);
```

如果类名保存在一个字符串中，这个字符串会在运行时变化，就可以使用这个方法在运行时获得不同的`Class`对象。如果`className`是一个类名或接口名，这个方法可以正常执行；否则，将抛出一个检查型异常（checked exception）。无论何时使用这个方法，都应该提供一个异常处理器。

第三种获得Class对象的方法是，如果`T`是任意的Java类型，`T.class`将代表匹配的`Class`对象。例如：

```
Class cl1 = Random.class; // import java.util.*
Class cl2 = int.class;
Class cl3 = Double[].class;
```

虚拟机为每个类型管理一个唯一的`Class`对象，因此，可以利用`==`运算符实现两个`Class`对象的比较。例如：

```
if (e.getClass() == Employee.getClass()) ...
```

如果有一个`Class`对象，可以用它构造类的实例。调用`getConstructor`方法将得到一个`Constructor`类型的对象，然后使用`newInstance`方法来构造一个实例。例如：

```
String className = "java.util.Random";
Class cl = Class.forName(className);
Object obj = cl.getConstructor().newInstance();
```

这两个方法的签名为：

```
/* java.lang.Class */
Constructor getConstructor(Class... parameterTypes)
    // 生成一个对象，描述有指定参数类型的构造器。如果这个类没有参数匹配的构造器，会抛出异常

/* java.lang.reflect.Constructor */
Object newInstance(Object... params)
    // 将 params 传递到构造器，来构造这个构造器声明类的一个新实例
```

2 声明异常入门

上面提到`forName`方法可能抛出异常，调用这个方法时要提供异常处理器。下面简要介绍一些异常处理的内容。

当运行时发生错误时，程序就会抛出一个异常，可以提供一个处理器捕获这个异常并进行处理。如果没有提供处理器，程序就会终止，并在控制台上打印出一个消息，给出异常的类型。

异常有两种类型：非检查型异常和检查型异常。对于检查型异常，这类异常难以避免，编译器将会检查这类异常，在编译阶段提示程序员解决问题。非检查型异常不检查，在编译阶段不报错，一般可以避免。例如，数组越界和访问`null`引用属于非检查型异常。

如果一个方法包含可能抛出检查型异常的语句，则在方法名上添加一个`throws`子句，调用这个方法的任何方法也都需要一个`throws`声明。例如：

```
public static void doSomethingWithClass(String name)
        throws ReflectiveOperationException
{
    Class cl = Class.forName(name); // 可能抛出异常
    // do something with cl
}
```

如果调用了一个可能抛出检查型异常的方法而没有提供相应的异常处理器，编译器就会报错。

3 利用反射分析类的能力

在`java.lang.reflect`包中有三个类`Field`、`Method`和`Constructor`，分别用于描述类的字段、方法和构造器。这三个类都有`getName`方法，用来返回字段、方法或构造器的名称：

```
String getName()
```

`Field`类有一个`getType`方法，用来返回描述字段类型的一个`Class`对象：

```
Class getType()
```

`Method`和`Constructor`类中的`getParameterTypes`方法返回参数类型：

```
Class[] getParameterTypes()
```

`Method`类中的`getReturnType`方法得到返回类型：

```
Class getReturnType()
```

这三个类都有一个`getModifier`方法，它返回一个整数，描述这个构造器、方法或字段的修饰符。可以使用`Modifier`类中的方法来分析这个整数值。

```
/* java.lang.reflect.Field/Method/Constructor */
int getModifier()

/* java.lang.reflect.Modifier */
static String toString(int modifiers)
    // 返回一个字符串，包含 modifiers 中对应的修饰符
static boolean isAbstract(int modifiers)
static boolean isFinal(int modifiers)
static boolean isInterface(int modifiers)
static boolean isNative(int modifiers)
static boolean isPrivate(int modifiers)
static boolean isProtected(int modifiers)
static boolean isPublic(int modifiers)
static boolean isStatic(int modifiers)
static boolean isStrict(int modifiers)
static boolean isSynchronized(int modifiers)
static boolean isVolatile(int modifiers)
    // 判断整数是否包含对应的修饰符
```

`Class`类中提供了一些返回`Field`、`Method`和`Constructor`对象的方法：

```
/* java.lang.Class */
Field getField(String name) // 得到指定名称的公共字段
Field[] getFields() // 返回这个类及其超类的公共字段
Field getDeclaredField(String name) // 得到指定名称的字段
Field[] getDeclaredFields() // 返回这个类的所有字段，不包括超类的字段
Method getMethod(String name, Class... parameterTypes) // 返回这个类中指定名称和参数类型的公共方法
Method[] getMethods() // 返回所有的公共方法，包括从超类继承来的公共方法
Method getDeclaredMethod(String name, Class... parameterTypes) // 返回这个类中指定名称和参数类型的方法
Method[] getDeclaredMethods() // 返回所有方法，但不包括从超类继承来的方法
Constructor[] getConstructors() // 返回这个类的所有公共构造器
Constructor[] getDeclaredConstructors() // 返回这个类的所有构造器
```

下面的例子展示了如何打印一个类的全部信息：

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

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

```
            throws ReflectiveOperationException
    {
        // 读入类名
        String name;
        if (args.length > 0)
            name = args[0];
        else
        {
            var in = new Scanner(System.in);
            System.out.println("Enter class name (e.g. java.util.Date): ");
            name = in.text();
        }

        // 输出类名和超类名
        Class cl = Class.forName(name);
        Class supercl = cl.getSuperclass();
        String modifiers = Modifier.toString(cl.getModifiers());
        if (modifiers.length() > 0)
            System.out.print(modifiers + " ");
        System.out.print("class " + name);
        if (supercl != null && supercl != Object.class)
            System.out.print(" extends " + supercl.getName());

        System.out.print("\n{\n");
        printConstructors(cl);
        System.out.println();
        printMethods(cl);
        System.out.println();
        printFields(cl);
        System.out.println("}");
    }

    public static void printConstructors(Class cl)
    {
        Constructor[] constructors = cl.getDeclaredConstructors();

        for (Constructor c : constructors)
        {
            String name = c.getName();
            System.out.print("  ");
            String modifiers = Modifier.toString(c.getModifiers());
            if (modifiers.length() > 0) System.out.print(modifiers + " ");
            System.out.print(name + "(");

            // 输出参数类型
            Class[] paramTypes = c.getParameterTypes();
            for (int j = 0; j < paramTypes,length; j++)
            {
                if (j > 0) System.out.print(", ");
                System,out.print(paramTypes[j].getName());
            }
            System.out.println(");");
        }
    }

    public static void printMethods(Class cl)
    {
        Method[] methods = cl.getDeclaredMethods();
```

```
        for (Method m : methods)
        {
            Class retType = m.getReturnType();
            String name = m.getName();

            System.out.print("  ");
            String modifiers = Modifier.toString(m.getModifiers());
            if (modifiers.length() > 0) System.out.print(modifiers + " ");
            System.out.print(retType.getName() + " " + name + "(");

            // 输出参数类型
            Class[] paramTypes = m.getParameterTypes();
            for (int j = 0; j < paramTypes,length; j++)
            {
                if (j > 0) System.out.print(", ");
                System,out.print(paramTypes[j].getName());
            }
            System.out.println(");");
        }
    }

    public static void printFields(Class cl)
    {
        Field[] fields = cl.getDeclaredFields();

        for (Field f : fields)
        {
            Class type = f.getType();
            String name = f.getName();
            Sysyem.out.print("  ")
            String modifiers = Modifier.toString(f.getModifiers());
            if (modifiers.length() > 0) System.out.print(modifiers + " ");
            System.out.println(type.getName() + " " + name + ";");
        }
    }
}
```

4 使用反射在运行时分析对象

利用反射机制可以查看在编译时还不知道的对象字段。`Field`类中的`get`方法可以得到字段的当前值，它的签名为：

```
Object get(Object obj) // 返回 obj 对象中用这个 Field 对象描述的字段的值
```

如果`obj`是一个对象，`f`是`obj`对象中某个字段对应的`Field`对象，那么`f.get(obj)`将返回对象`obj`的`f`字段的当前值。例如：

```
Employee harry = new Employee("Harry", 50000, 10, 1, 1989);
Class cl = harry.getClass(); // 得到 Employee 类
Field f = cl.getDeclaredField("name"); // 得到 Employee 类的 name 字段
Object v = f.get(harry); // 得到对象 harry 的 name 字段的当前值
```

`Field`类中的`set`方法可以修改字段的值，用法与`get`方法类似。它的签名为：

```
void set(Object obj, Object newValue) // 将 obj 对象中这个 Field 对象描述的字段设置为一个新值
```

只能对可以访问的字段使用`get`和`set`方法，否则会抛出`IllegalAccessException`异常。可见反射机制的默认行为受限于Java的访问控制。`AccessibleObject`类提供了修改访问控制的方法，从而打破这种限制。`AccessibleObject`类是`Field`、`Method`和`Constructor`类的公共超类，可以通过`Field`对象调用这些方法。

```
/* java.lang.reflect.AccessibleObject */
void setAeecssible(boolean flag)
    // 设置或取消这个可访问对象的可访问标志，如果拒绝访问则抛出 IllegalAccessException 异常
    // 设置为 true 表示抑制 Java 语言访问检查，false 表示强制执行 Java 语言访问检查
boolean trySetAccessible()
    // Java 9 引入，为这个可访问对象设置可访问标志，如果拒绝访问则返回 false
boolean isAccessible()
    // 得到这个可访问对象的可访问标志值
static void setAccessible(AccessibleObject[] array, boolean flag)
    // 便利方法，用于设置一个对象数组的可访问标志
```

下面的例子实现了一个可用于任意类的通用`toString`方法：

```
import java.lang.reflect.AccessibleObject;
import java.lang.reflect.Array;
import java.lang.reflect.Field;
import java.lang.reflect.Modifier;
import java.util.ArrayList;

public class ObjectAnalyzer
{
    private ArrayList<Object> visited = new ArrayList<>();

    public String toString(Object obj)
            throws ReflectiveOperationException
    {
        if (obj == null) return "null";
        if (visited.contains(obj)) return "..."; // 如果列表包含 obj 对象，说明 obj 已被访问过

        visited.add(obj);
        Class cl = obj.getClass();
        if (cl == String,class) return (String) obj; // 如果 obj 是 String

        if(cl.isArray()) //如果 obj 是数组
        {
            String r = cl.getComponentType().getName() + "[]{";
            for (int i = 0; i < Array.getLength(obj); i++)
            {
                if (i > 0) r += ",";
                Object val = Array.get(obj, i);
                if (cl.getComponentType().isPrimitive())
                    r += val;
                else
                    r += toString(val); // 递归
            }
            return r + "}";
```

```
        }

        String r = cl.getName();
        do
        {
            r += "[";
            Field[] fields = cl.getDeclaredFields();
            AccessibleObject.setAccessible(fields, true);
            for (Field f : fields)
            {
                if (!Modifier.isStatic(f.getModifiers()))
                {
                    if (!r.endsWith("[")) r += ",";
                    r += f.getName() + "=";
                    Class t = f.getType();
                    Object val = f.get(obj);
                    if (t.isPrimitive())
                        r += val;
                    else
                        r += toString(val); // 递归
                }
            }
            r += "]";
            cl = cl.getSuperclass();
        }
        while (cl != null);
        return r;
    }
}
```

上面的例子中用到的方法有：

```
/* java.util.ArrayList<E> */
boolean contains(Object o) // 如果列表包含指定的元素，则返回 true

/* java.lang.Class */
boolean isArray() // 判断此类对象是否表示数组类
Class getComponentType() // 返回数组类的元素类型对应的 Class 对象。如果此类不表示数组类，则返回 null
boolean isPrimitive() // 判断此类是否表示基本类型
Class getSuperclass() // 返回此类的超类对应的 Class 对象
```

5 使用反射编写泛型数组代码

`java.lang.reflect`包中的`Array`类允许动态创建和访问数组，其中的方法如下：

```
/* java.lang.reflect.Array */
static Object get(Object array, int index)
static boolean getBoolean(Object array, int index)
static byte getByte(Object array, int index)
static char getChar(Object array, int index)
static double getDouble(Object array, int index)
static float getFloat(Object array, int index)
static int getInt(Object array, int index)
static long getLong(Object array, int index)
static short getShort(Object array, int index)
```

```
    // 返回指定数组对象中指定索引位置上的值
static int getLength(Object array)
    // 返回指定数组的长度
static void set(Object array, int index, Object newValue)
static void setBoolean(Object array, int index, boolean newValue)
static void setByte(Object array, int index, byte newValue)
static void setChar(Object array, int index, char newValue)
static void setDouble(Object array, int index, double newValue)
static void setFloat(Object array, int index, float newValue)
static void setInt(Object array, int index, int newValue)
static void setLong(Object array, int index, long newValue)
static void setShort(Object array, int index, short newValue)
    // 将一个新值存储到指定数组对象中的指定索引位置上
static Object newInstance(Class componentType, int length)
static Object newInstance(Class componentType, int[] lengths)
    // 创建具有指定元素类型和大小的新数组
```

利用`Array`类和反射机制可以实现`Arrays`类中的`copyOf`方法，如下所示：

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

public class CopyOfTest
{
    public static void main(String[] args)
    {
        int[] a = {1, 2, 3};
        a = (int[]) goodCopyOf(a, 10);
        System.out.println(Arrays.toString(a));
    }

    public static Object goodCopyOf(Object a, int newLength)
    {
        Class cl = a.getClass();
        if (!cl.isArray()) return null;
        Class componentType = cl.getComponentType();
        int length = Array.getLength(a);
        Object newArray = Array.newInstance(componentType, newLength);
        System.arraycopy(a, 0, newArray, 0, Math.min(length, newLength));
        return newArray;
    }
}
```

6 调用任意方法

`Method`类中有一个`invoke`方法，用于调用包装在当前`Method`对象中的方法。它的签名为：

```
Object invoke(Object implicitParameter, Object[] explicitParameters)
    // 第一个参数是隐式参数，后面的参数是显式参数
    // 调用这个 Method 对象描述的方法，传入给定参数，返回方法的返回值。
    // 对于静态方法，第一个参数可以设置为 null
```

下面的例子打印一个数学函数的值的表格：

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

```
public class MethodTableTest
{
    public static void main(String[] args)
            throws ReflectiveOperationException
    {
        Method sqrt = Math.class.getMethod("sqrt", double.class);
        printTable(1, 10, 10, sqrt);
    }

    public static void printTable(double from, double to, int n, Method f)
    {
        System.out.println(f); // 输出方法名作为表头
        double dx = (to - from) / (n - 1);
        for (double x = from; x <= to; x += dx)
        {
            double y = (Double) f.invoke(null, x);
            System.out.printf("%10.4f | %10.4f\n", x, y);
        }
    }
}
```

这种调用方法的方式有很多缺点：

1. 通过`invoke`方法调用其他方法时，如果提供了错误的参数，就会抛出异常。这种编程风格不简便，而且很容易出错。
2. `invoke`方法的参数和返回值都是`Object`类型，这就意味着必须进行多次强制类型转换，这样编译器会丧失检查代码的机会，以至于到了测试阶段才会发现错误。
3. 使用反射获得方法指针的代码比直接调用方法慢得多。

因此，非必要时尽量不要使用这种方式。