04 反射

什么是反射？

反射就是Reflection，Java的反射是指程序在运行期可以拿到一个对象的所有信息。

正常情况下，如果我们要调用一个对象的方法，或者访问一个对象的字段，通常会传入对象实例：

```
// Main.java
import com.itranswarp.learnjava.Person;

public class Main {
    String getFullName(Person p) {
        return p.getFirstName() + " " + p.getLastName();
    }
}
```

但是，如果不能获得`Person`类，只有一个`Object`实例，比如这样：

```
String getFullName(Object obj) {
    return ???
}
```

怎么办？有童鞋会说：强制转型啊！

```
String getFullName(Object obj) {
    Person p = (Person) obj;
    return p.getFirstName() + " " + p.getLastName();
}
```

强制转型的时候，你会发现一个问题：编译上面的代码，仍然需要引用`Person`类。不然，去掉`import`语句，你看能不能编译通过？

所以，反射是为了解决在运行期，对某个实例一无所知的情况下，如何调用其方法。

Class类

除了`int`等基本类型外，Java的其他类型全部都是`class`（包括`interface`）。例如：

- `String`
- `Object`
- `Runnable`
- `Exception`
- ...

仔细思考，我们可以得出结论：`class`（包括`interface`）的本质是数据类型（`Type`）。无继承关系的数据类型无法赋值：

```
Number n = new Double(123.456); // OK
String s = new Double(123.456); // compile error!
```

而`class`是由JVM在执行过程中动态加载的。JVM在第一次读取到一种`class`类型时，将其加载进内存。

每加载一种`class`，JVM就为其创建一个`Class`类型的实例，并关联起来。注意：这里的`Class`类型是一个名叫`Class`的`class`。它长这样：

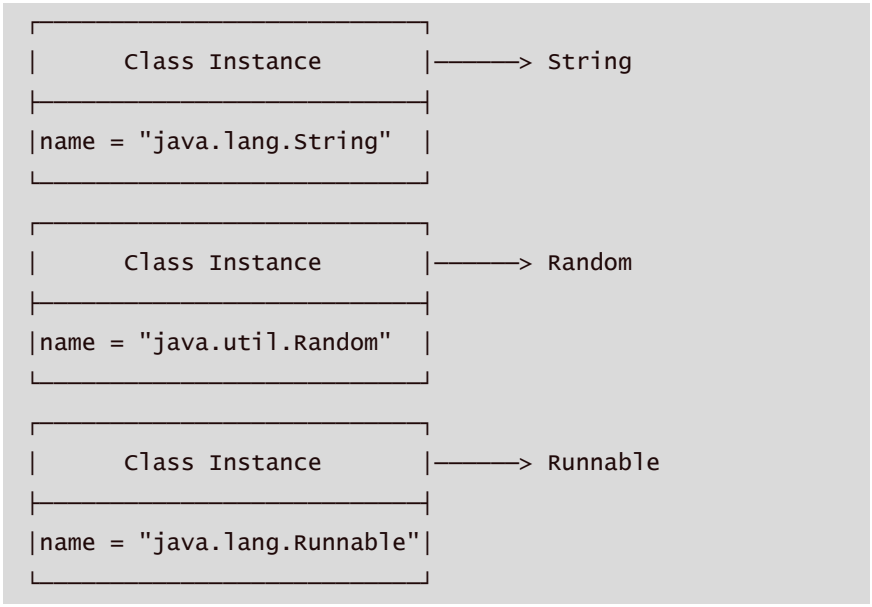
```
public final class Class {
    private Class() {}
}
```

以`String`类为例，当JVM加载`String`类时，它首先读取`String.class`文件到内存，然后，为`String`类创建一个`Class`实例并关联起来：

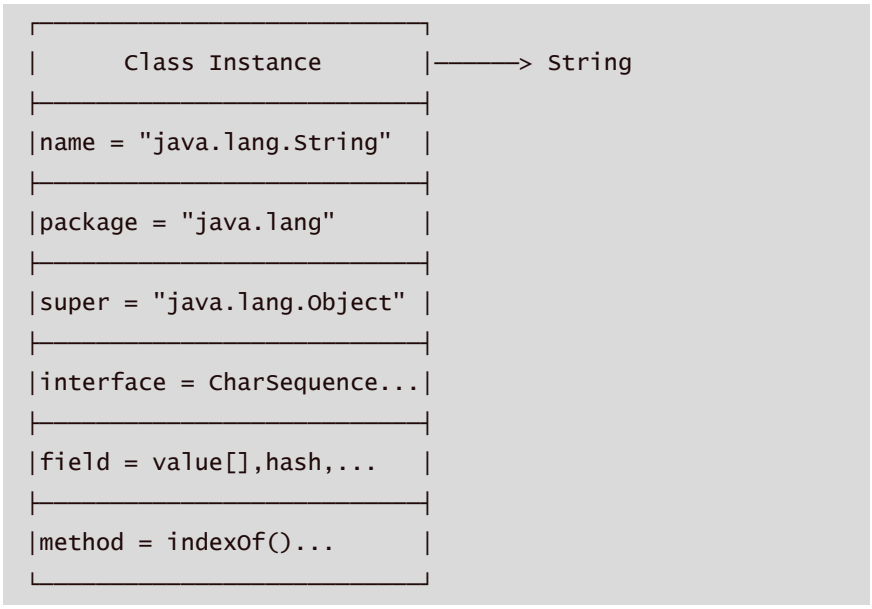
```
Class cls = new Class(String);
```

这个`Class`实例是JVM内部创建的，如果我们查看JDK源码，可以发现`Class`类的构造方法是`private`，只有JVM能创建`Class`实例，我们自己的Java程序是无法创建`Class`实例的。

所以，JVM持有的每个`Class`实例都指向一个数据类型（`class`或`interface`）：



一个`Class`实例包含了该`class`的所有完整信息：



由于JVM为每个加载的`class`创建了对应的`Class`实例，并在实例中保存了该`class`的所有信息，包括类名、包名、父类、实现的接口、所有方法、字段等，因此，如果获取了某个`Class`实例，我们就可以通过这个`Class`实例获取到该实例对应的`class`的所有信息。

这种通过`Class`实例获取`class`信息的方法称为反射（Reflection）。

如何获取一个`class`的`Class`实例？有三个方法：

方法一：直接通过一个`class`的静态变量`class`获取：

```
Class cls = String.class;
```

方法二：如果我们有一个实例变量，可以通过该实例变量提供的`getClass()`方法获取：

```
String s = "Hello";
Class cls = s.getClass();
```

方法三：如果知道一个`class`的完整类名，可以通过静态方法`Class.forName()`获取：

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

因为`Class`实例在JVM中是唯一的，所以，上述方法获取的`Class`实例是同一个实例。可以用`==`比较两个`Class`实例：

```
Class cls1 = String.class;

String s = "Hello";
Class cls2 = s.getClass();

boolean sameClass = cls1 == cls2; // true
```

注意一下`Class`实例比较和`instanceof`的差别：

```
Integer n = new Integer(123);

boolean b1 = n instanceof Integer; // true，因为n是Integer类型
boolean b2 = n instanceof Number; // true，因为n是Number类型的子类java

boolean b3 = n.getClass() == Integer.class; // true，因为n.getClass()返回Integer.class
boolean b4 = n.getClass() == Number.class; // false，因为Integer.class!=Number.class
```

用`instanceof`不但匹配指定类型，还匹配指定类型的子类。而用`==`判断`class`实例可以精确地判断数据类型，但不能作子类型比较。

通常情况下，我们应该用`instanceof`判断数据类型，因为面向抽象编程的时候，我们不关心具体的子类型。只有在需要精确判断一个类型是不是某个`class`的时候，我们才使用`==`判断`class`实例。

因为反射的目的是为了获得某个实例的信息。因此，当我们拿到某个`Object`实例时，我们可以通过反射获取该`Object`的`class`信息：

```
void printObjectInfo(Object obj) {
    Class cls = obj.getClass();
}
```

要从`Class`实例获取获取的基本信息，参考下面的代码：

```

```

```
public class Main {
    public static void main(String[] args) {
        printClassInfo("".getClass());
        printClassInfo(Runnable.class);
        printClassInfo(java.time.Month.class);
        printClassInfo(String[].class);
        printClassInfo(int.class);
    }

    static void printClassInfo(Class cls) {
        System.out.println("Class name: " +
cls.getName());
        System.out.println("Simple name: " +
cls.getSimpleName());
        if (cls.getPackage() != null) {
            System.out.println("Package name: " +
cls.getPackage().getName());
        }
        System.out.println("is interface: " +
cls.isInterface());
        System.out.println("is enum: " + cls.isEnum());
        System.out.println("is array: " + cls.isArray());
        System.out.println("is primitive: " +
cls.isPrimitive());
    }
}
```

注意到数组（例如`String[]`）也是一种`Class`，而且不同于`String.class`，它的类名是`[Ljava.lang.String`。此外，JVM为每一种基本类型如int也创建了`Class`，通过`int.class`访问。

如果获取到了一个`Class`实例，我们就可以通过该`Class`实例来创建对应类型的实例：

```
// 获取String的Class实例:
Class cls = String.class;
// 创建一个String实例:
String s = (String) cls.newInstance();
```

上述代码相当于`new String()`。通过`Class.newInstance()`可以创建类实例，它的局限是：只能调用`public`的无参数构造方法。带参数的构造方法，或者非`public`的构造方法都无法通过`Class.newInstance()`被调用。

动态加载

JVM在执行Java程序的时候，并不是一次性把所有用到的class全部加载到内存，而是第一次需要用到class时才加载。例如：

```
// Main.java
public class Main {
    public static void main(String[] args) {
        if (args.length > 0) {
            create(args[0]);
        }
    }

    static void create(String name) {
        Person p = new Person(name);
    }
}
```

当执行`Main.java`时，由于用到了`Main`，因此，JVM首先会把`Main.class`加载到内存。然而，并不会加载`Person.class`，除非程序执行到`create()`方法，JVM发现需要加载`Person`类时，才会首次加载`Person.class`。如果没有执行`create()`方法，那么`Person.class`根本就不会被加载。

这就是JVM动态加载`class`的特性。

动态加载`class`的特性对于Java程序非常重要。利用JVM动态加载`class`的特性，我们才能在运行期根据条件加载不同的实现类。例如，Commons Logging总是优先使用Log4j，只有当Log4j不存在时，才使用JDK的logging。利用JVM动态加载特性，大致的实现代码如下：

```
// Commons Logging优先使用Log4j:
LogFactory factory = null;
if (isClassPresent("org.apache.logging.log4j.Logger")) {
    factory = createLog4j();
} else {
    factory = createJdkLog();
}

boolean isClassPresent(String name) {
    try {
        Class.forName(name);
        return true;
    } catch (Exception e) {
        return false;
    }
}
```

这就是为什么我们只需要把Log4j的jar包放到classpath中，Commons Logging就会自动使用Log4j的原因。

小结

- JVM为每个加载的`class`及`interface`创建了对应的`Class`实例来保存`class`及`interface`的所有信息；

- 获取一个`class`对应的`Class`实例后，就可以获取该`class`的所有信息；
- 通过Class实例获取`class`信息的方法称为反射（Reflection）；
- JVM总是动态加载`class`，可以在运行期根据条件来控制加载class。

访问字段

对任意的一个`Object`实例，只要我们获取了它的`Class`，就可以获取它的一切信息。

我们先看看如何通过`Class`实例获取字段信息。`Class`类提供了以下几个方法来获取字段：

- Field getField(name)：根据字段名获取某个public的field（包括父类）
- Field getDeclaredField(name)：根据字段名获取当前类的某个field（不包括父类）
- Field[] getFields()：获取所有public的field（包括父类）
- Field[] getDeclaredFields()：获取当前类的所有field（不包括父类）

我们来看一下示例代码：

```
public class Main {
    public static void main(String[] args) throws
Exception {
        Class stdClass = Student.class;
        // 获取public字段"score":
        System.out.println(stdClass.getField("score"));
        // 获取继承的public字段"name":
        System.out.println(stdClass.getField("name"));
        // 获取private字段"grade":

 System.out.println(stdClass.getDeclaredField("grade"));
    }
}

class Student extends Person {
    public int score;
    private int grade;
}

class Person {
    public String name;
}
```

上述代码首先获取`Student`的`Class`实例，然后，分别获取`public`字段、继承的`public`字段以及`private`字段，打印出的`Field`类似：

```
public int Student.score
public java.lang.String Person.name
private int Student.grade
```

一个`Field`对象包含了一个字段的所有信息：

- `getName()`：返回字段名称，例如，`"name"`；
- `getType()`：返回字段类型，也是一个`Class`实例，例如，`String.class`；
- `getModifiers()`：返回字段的修饰符，它是一个`int`，不同的bit表示不同的含义。

以`String`类的`value`字段为例，它的定义是：

```
public final class String {
    private final byte[] value;
}
```

我们用反射获取该字段的信息，代码如下：

```
Field f = String.class.getDeclaredField("value");
f.getName(); // "value"
f.getType(); // class [B 表示byte[]类型
int m = f.getModifiers();
Modifier.isFinal(m); // true
Modifier.isPublic(m); // false
Modifier.isProtected(m); // false
Modifier.isPrivate(m); // true
Modifier.isStatic(m); // false
```

获取字段值

利用反射拿到字段的一个`Field`实例只是第一步，我们还可以拿到一个实例对应的该字段的值。

例如，对于一个`Person`实例，我们可以先拿到`name`字段对应的`Field`，再获取这个实例的`name`字段的值：

```
// reflection
import java.lang.reflect.Field;
public class Main {

    public static void main(String[] args) throws
Exception {
        Object p = new Person("Xiao Ming");
        Class c = p.getClass();
        Field f = c.getDeclaredField("name");
        Object value = f.get(p);
        System.out.println(value); // "Xiao Ming"
    }
}

class Person {
    private String name;
```

```
    public Person(String name) {
        this.name = name;
    }
}
```

上述代码先获取`Class`实例，再获取`Field`实例，然后，用`Field.get(Object)`获取指定实例的指定字段的值。

运行代码，如果不出意外，会得到一个`IllegalAccessException`，这是因为`name`被定义为一个`private`字段，正常情况下，`Main`类无法访问`Person`类的`private`字段。要修复错误，可以将`private`改为`public`，或者，在调用`Object value = f.get(p);`前，先写一句：

```
f.setAccessible(true);
```

调用`Field.setAccessible(true)`的意思是，别管这个字段是不是`public`，一律允许访问。

可以试着加上上述语句，再运行代码，就可以打印出`private`字段的值。

有童鞋会问：如果使用反射可以获取`private`字段的值，那么类的封装还有什么意义？

答案是正常情况下，我们总是通过`p.name`来访问`Person`的`name`字段，编译器会根据`public`、`protected`和`private`决定是否允许访问字段，这样就达到了数据封装的目的。

而反射是一种非常规的用法，使用反射，首先代码非常繁琐，其次，它更多地是给工具或者底层框架来使用，目的是在不知道目标实例任何信息的情况下，获取特定字段的值。

此外，`setAccessible(true)`可能会失败。如果JVM运行期存在`SecurityManager`，那么它会根据规则进行检查，有可能阻止`setAccessible(true)`。例如，某个`SecurityManager`可能不允许对`java`和`javax`开头的`package`的类调用`setAccessible(true)`，这样可以保证JVM核心库的安全。

设置字段值

通过Field实例既然可以获取到指定实例的字段值，自然也可以设置字段的值。

设置字段值是通过`Field.set(Object, Object)`实现的，其中第一个`Object`参数是指定的实例，第二个`Object`参数是待修改的值。示例代码如下：

```
// reflection
import java.lang.reflect.Field;
public class Main {

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

```
        Person p = new Person("Xiao Ming");
        System.out.println(p.getName()); // "Xiao Ming"
        Class c = p.getClass();
        Field f = c.getDeclaredField("name");
        f.setAccessible(true);
        f.set(p, "Xiao Hong");
        System.out.println(p.getName()); // "Xiao Hong"
    }
}

class Person {
    private String name;

    public Person(String name) {
        this.name = name;
    }

    public String getName() {
        return this.name;
    }
}
```

运行上述代码，打印的`name`字段从`Xiao Ming`变成了`Xiao Hong`，说明通过反射可以直接修改字段的值。

同样的，修改非`public`字段，需要首先调用`setAccessible(true)`。

练习

利用反射给字段赋值：下载练习：reflect-field （推荐使用IDE练习插件快速下载）

小结

- Java的反射API提供的`Field`类封装了字段的所有信息：
- 通过`Class`实例的方法可以获取`Field`实例：`getField()`，`getFields()`，`getDeclaredField()`，`getDeclaredFields()`；
- 通过Field实例可以获取字段信息：`getName()`，`getType()`，`getModifiers()`；
- 通过Field实例可以读取或设置某个对象的字段，如果存在访问限制，要首先调用`setAccessible(true)`来访问非`public`字段。
- 通过反射读写字段是一种非常规方法，它会破坏对象的封装。

调用方法

我们已经能通过`Class`实例获取所有`Field`对象，同样的，可以通过`Class`实例获取所有`Method`信息。`Class`类提供了以下几个方法来获取`Method`：

- `Method getMethod(name, Class...)`：获取某个`public`的`Method`（包括父类）

- `Method getDeclaredMethod(name, Class...)`：获取当前类的某个`Method`（不包括父类）
- `Method[] getMethods()`：获取所有`public`的`Method`（包括父类）
- `Method[] getDeclaredMethods()`：获取当前类的所有`Method`（不包括父类）

我们来看一下示例代码：

```
public class Main {
    public static void main(String[] args) throws
Exception {
        Class stdClass = Student.class;
        // 获取public方法getScore，参数为String:
        System.out.println(stdClass.getMethod("getScore",
String.class));
        // 获取继承的public方法getName，无参数:
        System.out.println(stdClass.getMethod("getName"));
        // 获取private方法getGrade，参数为int:

 System.out.println(stdClass.getDeclaredMethod("getGrade",
int.class));
    }
}

class Student extends Person {
    public int getScore(String type) {
        return 99;
    }
    private int getGrade(int year) {
        return 1;
    }
}

class Person {
    public String getName() {
        return "Person";
    }
}
```

上述代码首先获取`Student`的`Class`实例，然后，分别获取`public`方法、继承的`public`方法以及`private`方法，打印出的`Method`类似：

```
public int Student.getScore(java.lang.String)
public java.lang.String Person.getName()
private int Student.getGrade(int)
```

一个`Method`对象包含一个方法的所有信息：

- `getName()`：返回方法名称，例如：`"getScore"`；

- `getReturnType()`：返回方法返回值类型，也是一个Class实例，例如：`String.class`；
- `getParameterTypes()`：返回方法的参数类型，是一个Class数组，例如：`{String.class, int.class}`；
- `getModifiers()`：返回方法的修饰符，它是一个`int`，不同的bit表示不同的含义。

调用方法

当我们获取到一个`Method`对象时，就可以对它进行调用。我们以下面的代码为例：

```
String s = "Hello world";
String r = s.substring(6); // "world"
```

如果用反射来调用`substring`方法，需要以下代码：

```
// reflection
import java.lang.reflect.Method;
public class Main {
    public static void main(String[] args) throws
Exception {
        // String对象:
        String s = "Hello world";
        // 获取String substring(int)方法，参数为int:
        Method m = String.class.getMethod("substring",
int.class);
        // 在s对象上调用该方法并获取结果:
        String r = (String) m.invoke(s, 6);
        // 打印调用结果:
        System.out.println(r);
    }
}
```

注意到`substring()`有两个重载方法，我们获取的是`String substring(int)`这个方法。思考一下如何获取`String substring(int, int)`方法。

对`Method`实例调用`invoke`就相当于调用该方法，`invoke`的第一个参数是对象实例，即在哪个实例上调用该方法，后面的可变参数要与方法参数一致，否则将报错。

调用静态方法

如果获取到的Method表示一个静态方法，调用静态方法时，由于无需指定实例对象，所以`invoke`方法传入的第一个参数永远为`null`。我们以`Integer.parseInt(String)`为例：

```
// reflection
import java.lang.reflect.Method;
public class Main {
    public static void main(String[] args) throws
Exception {
        // 获取Integer.parseInt(String)方法，参数为String:
        Method m = Integer.class.getMethod("parseInt",
String.class);
        // 调用该静态方法并获取结果:
        Integer n = (Integer) m.invoke(null, "12345");
        // 打印调用结果:
        System.out.println(n);
    }
}
```

调用非public方法

和Field类似，对于非public方法，我们虽然可以通过`Class.getDeclaredMethod()`获取该方法实例，但直接对其调用将得到一个`IllegalAccessException`。为了调用非public方法，我们通过`Method.setAccessible(true)`允许其调用：

```
// reflection
import java.lang.reflect.Method;
public class Main {
    public static void main(String[] args) throws
Exception {
        Person p = new Person();
        Method m =
p.getClass().getDeclaredMethod("setName", String.class);
        m.setAccessible(true);
        m.invoke(p, "Bob");
        System.out.println(p.name);
    }
}

class Person {
    String name;
    private void setName(String name) {
        this.name = name;
    }
}
```

此外，`setAccessible(true)`可能会失败。如果JVM运行期存在`SecurityManager`，那么它会根据规则进行检查，有可能阻止`setAccessible(true)`。例如，某个`SecurityManager`可能不允许对`java`和`javax`开头的`package`的类调用`setAccessible(true)`，这样可以保证JVM核心库的安全。

多态

我们来考察这样一种情况：一个`Person`类定义了`hello()`方法，并且它的子类`Student`也覆写了`hello()`方法，那么，从`Person.class`获取的`Method`，作用于`Student`实例时，调用的方法到底是哪个？

```
// reflection
import java.lang.reflect.Method;
public class Main {
    public static void main(String[] args) throws
Exception {
        // 获取Person的hello方法:
        Method h = Person.class.getMethod("hello");
        // 对Student实例调用hello方法:
        h.invoke(new Student());
    }
}

class Person {
    public void hello() {
        System.out.println("Person:hello");
    }
}

class Student extends Person {
    public void hello() {
        System.out.println("Student:hello");
    }
}
```

运行上述代码，发现打印出的是`Student:hello`，因此，使用反射调用方法时，仍然遵循多态原则：即总是调用实际类型的覆写方法（如果存在）。上述的反射代码：

```
Method m = Person.class.getMethod("hello");
m.invoke(new Student());
```

实际上相当于：

```
Person p = new Student();
p.hello();
```

练习

利用反射调用方法：下载练习：reflect-method （推荐使用IDE练习插件快速下载）

小结

- Java的反射API提供的Method对象封装了方法的所有信息：
- 通过`Class`实例的方法可以获取`Method`实例：`getMethod()`，`getMethods()`，`getDeclaredMethod()`，

`getDeclaredMethods()`；

- 通过`Method`实例可以获取方法信息：`getName()`，`getReturnType()`，`getParameterTypes()`，`getModifiers()`；
- 通过`Method`实例可以调用某个对象的方法：`Object invoke(Object instance, Object... parameters)`；
- 通过设置`setAccessible(true)`来访问非`public`方法；
- 通过反射调用方法时，仍然遵循多态原则。

调用构造方法

我们通常使用`new`操作符创建新的实例：

```
Person p = new Person();
```

如果通过反射来创建新的实例，可以调用Class提供的newInstance()方法：

```
Person p = Person.class.newInstance();
```

调用Class.newInstance()的局限是，它只能调用该类的public无参数构造方法。如果构造方法带有参数，或者不是public，就无法直接通过Class.newInstance()来调用。

为了调用任意的构造方法，Java的反射API提供了Constructor对象，它包含一个构造方法的所有信息，可以创建一个实例。Constructor对象和Method非常类似，不同之处仅在于它是一个构造方法，并且，调用结果总是返回实例：

```
import java.lang.reflect.Constructor;
public class Main {
    public static void main(String[] args) throws
Exception {
        // 获取构造方法Integer(int):
        Constructor cons1 =
Integer.class.getConstructor(int.class);
        // 调用构造方法:
        Integer n1 = (Integer) cons1.newInstance(123);
        System.out.println(n1);

        // 获取构造方法Integer(String)
        Constructor cons2 =
Integer.class.getConstructor(String.class);
        Integer n2 = (Integer) cons2.newInstance("456");
        System.out.println(n2);
    }
}
```

通过Class实例获取Constructor的方法如下：

- `getConstructor(Class...)`：获取某个`public`的`Constructor`；
- `getDeclaredConstructor(Class...)`：获取某个`Constructor`；
- `getConstructors()`：获取所有`public`的`Constructor`；

- `getDeclaredConstructors()`：获取所有`Constructor`。

注意`Constructor`总是当前类定义的构造方法，和父类无关，因此不存在多态的问题。

调用非`public`的`Constructor`时，必须首先通过`setAccessible(true)`设置允许访问。`setAccessible(true)`可能会失败。

小结

- `Constructor`对象封装了构造方法的所有信息；
- 通过`Class`实例的方法可以获取`Constructor`实例：`getConstructor()`，`getConstructors()`，`getDeclaredConstructor()`，`getDeclaredConstructors()`；
- 通过`Constructor`实例可以创建一个实例对象：`newInstance(Object... parameters)`；通过设置`setAccessible(true)`来访问非`public`构造方法。

获取继承关系

当我们获取到某个`Class`对象时，实际上就获取到了一个类的类型：

```
Class cls = String.class; // 获取到String的Class
```

还可以用实例的`getClass()`方法获取：

```
String s = "";
Class cls = s.getClass(); // s是String，因此获取到String的
Class
```

最后一种获取`Class`的方法是通过`Class.forName("")`，传入`Class`的完整类名获取：

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

这三种方式获取的`Class`实例都是同一个实例，因为JVM对每个加载的`Class`只创建一个`Class`实例来表示它的类型。

获取父类的Class

有了`Class`实例，我们还可以获取它的父类的`Class`：

```
public class Main {
    public static void main(String[] args) throws
Exception {
        Class i = Integer.class;
        Class n = i.getSuperclass();
        System.out.println(n);
        Class o = n.getSuperclass();
        System.out.println(o);
        System.out.println(o.getSuperclass());
    }
}
```

运行上述代码，可以看到，`Integer`的父类类型是`Number`，`Number`的父类是`Object`，`Object`的父类是`null`。除`Object`外，其他任何非`interface`的`Class`都必定存在一个父类类型。

获取interface

由于一个类可能实现一个或多个接口，通过`Class`我们就可以查询到实现的接口类型。例如，查询`Integer`实现的接口：

```
// reflection
import java.lang.reflect.Method;
public class Main {
    public static void main(String[] args) throws
Exception {
        Class s = Integer.class;
        Class[] is = s.getInterfaces();
        for (Class i : is) {
            System.out.println(i);
        }
    }
}
```

运行上述代码可知，`Integer`实现的接口有：

- java.lang.Comparable
- java.lang.constant.Constable
- java.lang.constant.ConstantDesc

要特别注意：`getInterfaces()`只返回当前类直接实现的接口类型，并不包括其父类实现的接口类型：

```
// reflection
import java.lang.reflect.Method;
public class Main {
    public static void main(String[] args) throws
Exception {
        Class s = Integer.class.getSuperclass();
        Class[] is = s.getInterfaces();
        for (Class i : is) {
            System.out.println(i);
        }
    }
}
```

`Integer`的父类是`Number`，`Number`实现的接口是`java.io.Serializable`。

此外，对所有`interface`的`Class`调用`getSuperclass()`返回的是`null`，获取接口的父接口要用`getInterfaces()`：

```
System.out.println(java.io.DataInputStream.class.getSuperc
lass()); // java.io.FilterInputStream，因为DataInputStream
继承自FilterInputStream
System.out.println(java.io.Closeable.class.getSuperclass()
); // null，对接口调用getSuperclass()总是返回null，获取接口的父
接口要用getInterfaces()
```

如果一个类没有实现任何`interface`，那么`getInterfaces()`返回空数组。

继承关系

当我们判断一个实例是否是某个类型时，正常情况下，使用`instanceof`操作符：

```
Object n = Integer.valueOf(123);
boolean isDouble = n instanceof Double; // false
boolean isInteger = n instanceof Integer; // true
boolean isNumber = n instanceof Number; // true
boolean isSerializable = n instanceof
java.io.Serializable; // true
```

如果是两个`Class`实例，要判断一个向上转型是否成立，可以调用`isAssignableFrom()`：

```
// Integer i = ?
Integer.class.isAssignableFrom(Integer.class); // true，因为Integer可以赋值给Integer
// Number n = ?
Number.class.isAssignableFrom(Integer.class); // true，因为Integer可以赋值给Number
// Object o = ?
Object.class.isAssignableFrom(Integer.class); // true，因为Integer可以赋值给Object
// Integer i = ?
Integer.class.isAssignableFrom(Number.class); // false，因为Number不能赋值给Integer
```

小结

通过`Class`对象可以获取继承关系：

- `Class getSuperclass()`：获取父类类型；
- `Class[] getInterfaces()`：获取当前类实现的所有接口。

通过`Class`对象的`isAssignableFrom()`方法可以判断一个向上转型是否可以实现。

动态代理

我们来比较Java的`class`和`interface`的区别：

- 可以实例化`class`（非`abstract`）；
- 不能实例化`interface`。

所有`interface`类型的变量总是通过向上转型并指向某个实例的：

```
CharSequence cs = new StringBuilder();
```

有没有可能不编写实现类，直接在运行期创建某个`interface`的实例呢？

这是可能的，因为Java标准库提供了一种动态代理（Dynamic Proxy）的机制：可以在运行期动态创建某个`interface`的实例。

什么叫运行期动态创建？听起来好像很复杂。所谓动态代理，是和静态相对应的。我们来看静态代码怎么写：

定义接口：

```
public interface Hello {
    void morning(String name);
}
```

编写实现类：

```
public class HelloWorld implements Hello {
    public void morning(String name) {
        System.out.println("Good morning, " + name);
    }
}
```

创建实例，转型为接口并调用：

```
Hello hello = new HelloWorld();
hello.morning("Bob");
```

这种方式就是我们通常编写代码的方式。

还有一种方式是动态代码，我们仍然先定义了接口`Hello`，但是我们并不去编写实现类，而是直接通过JDK提供的一个`Proxy.newProxyInstance()`创建了一个`Hello`接口对象。这种没有实现类但是在运行期动态创建了一个接口对象的方式，我们称为动态代码。JDK提供的动态创建接口对象的方式，就叫动态代理。

一个最简单的动态代理实现如下：

```
import java.lang.reflect.InvocationHandler;
import java.lang.reflect.Method;
import java.lang.reflect.Proxy;
public class Main {
    public static void main(String[] args) {
        InvocationHandler handler = new
InvocationHandler() {
            @Override
            public Object invoke(Object proxy, Method
method, Object[] args) throws Throwable {
                System.out.println(method);
                if (method.getName().equals("morning")) {
                    System.out.println("Good morning, " +
args[0]);
                }
                return null;
            }
        };
        Hello hello = (Hello) Proxy.newProxyInstance(
            Hello.class.getClassLoader(), // 传入
ClassLoader
            new Class[] { Hello.class }, // 传入要实现的接口
            handler); // 传入处理调用方法的InvocationHandler
        hello.morning("Bob");
    }
}

interface Hello {
    void morning(String name);
}
```

在运行期动态创建一个`interface`实例的方法如下：

1. 定义一个`InvocationHandler`实例，它负责实现接口的方法调用；
2. 通过

   ```
   Proxy.newProxyInstance()
   ```

   创建

   ```
   interface
   ```

   实例，它需要3个参数：

   a. 使用的`ClassLoader`，通常就是接口类的`ClassLoader`；
   b. 需要实现的接口数组，至少需要传入一个接口进去；
   c. 用来处理接口方法调用的`InvocationHandler`实例。
3. 将返回的`Object`强制转型为接口。

动态代理实际上是JDK在运行期动态创建class字节码并加载的过程，它并没有什么黑魔法，把上面的动态代理改写为静态实现类大概长这样：

```
public class HelloDynamicProxy implements Hello {
    InvocationHandler handler;
    public HelloDynamicProxy(InvocationHandler handler) {
        this.handler = handler;
    }
    public void morning(String name) {
        handler.invoke(
           this,
           Hello.class.getMethod("morning"),
           new Object[] { name });
    }
}
```

其实就是JDK帮我们自动编写了一个上述类（不需要源码，可以直接生成字节码），并不存在可以直接实例化接口的黑魔法。

小结

- Java标准库提供了动态代理功能，允许在运行期动态创建一个接口的实例；
- 动态代理是通过`Proxy`创建代理对象，然后将接口方法“代理”给`InvocationHandler`完成的。