Chp9　接口

本章导读

接口是 Java 语言中的核心概念之一。这个语法特性与“多态”具有非常紧密的联系。在学习接口之前，请读者首先复习一下多态的相关知识和练习，确认已经对多态有比较牢固的掌握之后，再进行下一步接口的学习。

1 接口的语法

1.1 接口是特殊的抽象类

从语法特性上说，接口很类似于抽象类。

如果有一个抽象类，其所有属性都是公开静态常量，所有方法都是公开抽象方法，例如下面代码所示。

```
abstract class MyAbstractClass{
    public static final int VALUE1 = 100; //属性是公开静态常量
    public static final int VALUE2 = 200; //第二个属性
    public abstract void m1(); //方法是公开抽象方法
    public abstract void m2(int n); //第二个方法
}
```

上面的 MyAbstractClass 类，具有的两个属性 VALUE1 和 VALUE2 都是公开静态常量，具有的两个方法 m1 和 m2 都是公开抽象方法。

由于 MyAbstractClass 是抽象类，因此无法创建对象，只能够声明引用。如果要创建对象的话，必须要写一个类继承 MyAbstractClass 类，并且实现这个类中的 m1 和 m2 方法。例如下面的代码：

```
class MySubClass extends MyAbstractClass{
    public void m1(){}
    public void m2(int n){}
}
```

要注意的是，方法覆盖要求“子类的访问修饰符相同或更宽”，由于 MyAbstarctClass 类中的 m1 方法和 m2 方法都是 public 的，MySubClass 中的 m1 和 m2 方法的访问修饰符也必须是 public 的。

对于 MyAbstractClass 这种特殊的抽象类，我们可以把其改写成接口。接口的特点和之前我们提到的 MyAbstractClass 的特点相同：

1、 所有属性都是公开静态常量

2、 所有方法都是公开抽象方法

使用关键字 interface 来定义接口。把 MyAbstractClass 改写成接口，结果如下：

```
interface MyInterface{
    public static final int VALUE1 = 100;
    public static final int VALUE2 = 200;
```

```
    public abstract void m1();
    public abstract void m2(int n);
}
```

注意，interface 替代了 abstract class 这两个关键字。interface 关键字和 class 关键字类似，一个接口编译后会生成一个.class 文件；一个.java 文件中可以有多个接口，但是最多只能有一个公开的接口，且公开接口的接口名与文件名相同。

既然接口中所有属性都是公开静态常量，则接口中的属性，可以省略 public static final 关键字；同样的，由于接口中所有方法都是公开抽象方法，因此可以省略 public abstract 关键字。因此，上面的 MyInterface 可以改写如下：

```
interface MyInterface{
    int VALUE1 = 100;
    int VALUE2 = 200;
    void m1();
    void m2(int n);
}
```

上面的这个接口中，虽然没有写 public static final，但是其属性都是公开静态常量；虽然没有写 public abstract，但是其方法都是公开抽象方法。

与抽象类类似，接口可以声明引用，但是不能创建对象。接口与抽象类不同的在于，抽象类中可以定义构造方法，以供子类的构造方法调用，而接口中不能定义任何构造方法，系统也不会提供默认无参的构造方法。

类似于子类继承抽象类，接口也可以被子类“继承”。只不过，接口具有自己的关键字：implements。使用这个关键字表示“实现”，类似于抽象类中子类继承父类的关系。例如，下面的 MyImpl 类就实现了 MyInterface 接口。

```
class MyImpl implements MyInterface{
    public void m1(){}
    public void m2(int n){}
}
```

要注意的是：

1. 一个类实现接口，如果不希望这个类作为抽象类，则应该实现接口中定义的所有方法。
2. 接口中所有的方法都是公开方法。因此，在实现接口中的方法时，实现类的方法也必须写成公开的！由于类中的方法，默认访问修饰符是“default”，因此，在实现接口中的方法时，修饰符“public”不能省略。

接口最基本的使用就介绍到这里，从这一节的内容可以看出，从语法上说，接口很类似特殊的抽象类，只不过在接口语法中增加了两个关键字：interface 和 implements 而已。

除此之外，抽象类之间可以继承，同样的，接口之间也可以继承。接口之间继承时，使用的关键字同样为 extends。例如：

```
interface IA{
    void ma();
}

interface IB extends IA{
    void mb();
}
```

上面这个例子中，IB 接口继承自 IA 接口。因此，IB 中存在两个方法：ma 方法是从 IA 接口中继承来的，mb 方法是 IB 接口自身定义的。如果有一个类要实现 IB 接口，则必须实现 ma 和 mb 两个方法，例如：

```
class IAIBImpl implements IB{
    public void ma(){}
    public void mb(){}
}
```

1.2 多继承

当然，接口和抽象类除了关键字不同外，还有一些非常重要的不同之处。

首先，接口之间可以多继承。不同于 Java 中对于类之间的单继承的要求，接口之间没有这个限制。一个接口可以继承多个接口，例如下面这个例子：

```
interface IA{
    void ma();
}
interface IB{
    void mb();
}
interface  IC extends  IA, IB{  //IC 同时继承 IA 和 IB 两个接口
    void mc();
}
```

上面这段代码中，IC 接口继承自 IA 和 IB 接口。注意，继承多个接口时，多个接口之间用逗号隔开。IC 同时继承这两个接口，因此 IC 中同时包含有这两个接口中定义的方法，并且包含自身定义的 mc 方法。因此，如果有一个类要实现 IC 接口的话，则需要实现三个方法：ma、mb 以及 mc。

除此之外，一个类在继承另外一个类的同时，还可以实现多个接口，例如下面的例子：

```
interface IA{
    void ma();
}
interface IB(){
    void mb();
}
//IC 继承自 IA，IB 接口。这里是接口的多继承
interface IC extends IA, IB{
    void mc();
}

interface ID{
    void md();
}
abstract class ClassE{
    public abstract void me();
```

```
}
/*
    一个类可以继承自一个类，并实现多个接口
    MyImpl 继承自 ClassE 类，实现了 IC 和 ID 接口
    注意，先写继承自哪个类，再写实现了哪些接口
*/
class MyImpl extends ClassE implements IC, ID{
    //IC 接口中包含 ma、mb、mc 方法
    public void ma(){}
    public void mb(){}
    public void mc(){}
    //ID 接口中包含 md 方法
    public void md(){}
    //ClassE 中包含 me 方法
    public void me(){}
}
```

以上就是接口和抽象类不同的地方：接口和接口之间可以多继承；一个类在继承一个父类的同时，还能够实现多个接口。

1.3 接口与多态

有了接口的多继承特性，加上一个类能够实现多个接口，这样，接口结合多态，语法和概念都变得非常的灵活。例如，在之前 MyImpl 类的基础上，写以下代码：

```
public class TestMyImpl{
    public static void main(String args[]){
        IA ia = new MyImpl();
        System.out.println(ia instanceof IA);
        System.out.println(ia instanceof IB);
        System.out.println(ia instanceof IC);
        System.out.println(ia instanceof ID);
        System.out.println(ia instanceof ClassE);
        System.out.println(ia instanceof MyImpl);
    }
}
```

上面的程序会输出 6 个 true。

我们以前讲过，instanceof 关键字用来判断对象和某个类型是否兼容。

在这段代码中，ia 引用指向了一个 MyImpl 类的对象。我们知道，MyImpl 类实现了 ID 接口，如果把 ID 接口看作是一个特殊的抽象类，那么 MyImpl 类就可以看作是这个抽象类的子类，因此代码

```
System.out.println(ia  instanceof  ID);
```

会输出“true”。

同理，MyImpl 类还实现了 IC 接口，而 IC 接口是 IA，IB 两个接口的子接口。因此，MyImpl 类也可看作是 IA，IB 两个接口的实现类，因此代码

```
System.out.println(ia instanceof IA);
System.out.println(ia instanceof IB);
```

```
System.out.println(ia instanceof IC);
```
也会输出“true”。

当然，MyImpl 类本身又继承了 ClassE 类，因此代码：

```
System.out.println(ia instanceof ClassE);
System.out.println(ia instanceof MyImpl);
```
也会输出“true”。

所以，上述代码会输出 6 个“true”。因此，根据多态的语法，一个 MyImpl 类的对象可以放入:IA,IB,IC,ID,ClassE,MyImpl 六种不同类型的引用中。

实际上，对于初学者，接口很多比较难掌握和理解的东西，都跟接口的多态特性有关。只要掌握好了多态，就能相当程度上把握住接口的应用。

例如下面这个例子：

首先定义一个 Teacher 接口：

```
interface Teacher{
    void teach();
}
```
然后，为接口提供两个实现类：

```
class CoreJavaTeacher implements Teacher{
    public void teach(){
        System.out.println(“teach corejava”);
    }
}
class JavaWebTeacher implements Teacher{
    public void teach(){
        System.out.println(“teach java web”);
    }
}
```
之后，提供一个 TestTeacher 类如下：

```
public class TestTeacher{
    public static void main(String args[]){
        Teacher t = getTeacher(0);
        beginClass(t);
    }

    public static Teacher getTeacher(int type){
        if(type == 0) return new CoreJavaTeacher();
        else return new JavaWebTeacher();
    }

    public static void beginClass(Teacher t){
        t.teach();
    }
}
```
在注意在上面的代码中的两个函数。首先，getTeacher 方法的返回值为一个 Teacher 对

象。由于 Teacher 类型是一个接口类型，因此不会返回一个真正的接口对象，而返回的对象一定是接口的某一个实现类的对象。这是把多态用在方法的返回值类型上。

而 beginClass 方法能够接受一个 Teacher 类型的参数。由于 Teacher 是接口类型，因此接受的参数一定是 Teacher 接口实现类的对象。这是把多态用在方法的参数类型上。

以上的代码与我们在多态章节中见到的代码有很多类似之处，不同的是，将“父类”的概念换成了“接口”的概念。在含义上，这二者是相同的。

2 接口的作用

上一部分我们介绍了接口的语法。事实上，接口的语法并不困难，甚至于应该说相当的简单。与接口的语法相比，更难以理解和掌握的，是接口的作用，以及如何更好的利用接口。

在 Java 中，接口主要用来实现两大功能：一是用接口实现多继承；二是用接口来进行解耦合。其中，后者是接口最重要的作用。

2.1 接口与多继承

在介绍 Java 的历史和背景时，我们曾经介绍过，开发 Java 语言的工程师，都有多年 Unix 下使用 C++语言进行开发的经验。而且，Java 中有很多的语法以及关键字类似于 C++，可以说，Java 语言是一种脱胎于 C++的语言。

虽然有些语法类似，但是 Java 语言与 C++语言有着本质的区别。Java 语言和 C++最大的区别之一，就是多继承：C++支持多继承，而 Java 语言只允许单继承。

例如，假设我们设计了一个类：Spider，这个类表示蜘蛛。又设计了一个类 Man，这个类表示人。现在，我们要设计一个类：SpiderMan，表示“蜘蛛侠”。

很显然，“蜘蛛侠”这个类，既有 Man 这个类的特点（会说话，会走路，会恋爱……），也有 Spider 这个类的特点（能射出蛛丝）。也就是说，我们可以把蜘蛛侠当做特殊的蜘蛛，也可以把蜘蛛侠当做特殊的人。从这个意义上说，SpiderMan 应当既是 Spider 的子类，又是 Man 的子类。

C++语言支持多继承，因此在 C++语言中，我们可以让 SpiderMan 这个类直接继承自两个父类。

但是，在 Java 中只允许单继承。为此，我们只能为 SpiderMan 选择唯一的一个父类，并让其实现别的接口。例如，我们可以让 SpiderMan 继承自 Man 类，然后，把 Spider 作为接口的形式，让 SpiderMan 实现这个接口。

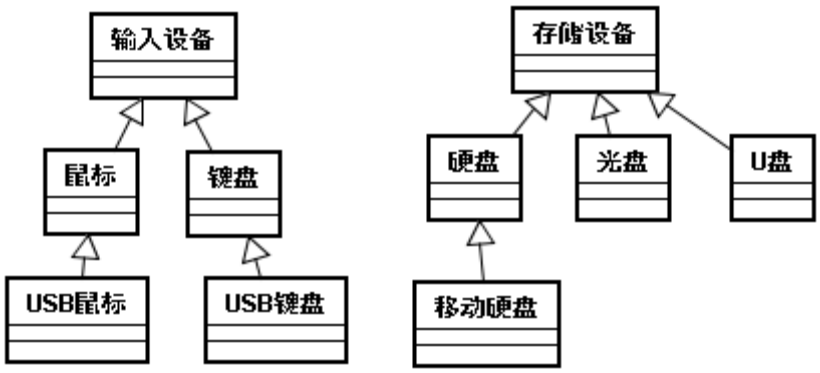
```
class SpiderMan extends Man implements Spider
```

虽然 Java 中只允许单继承，但是对于实现接口，Java 没有做数量上的限制。例如，SpiderMan 这个类只有一个父类 Man。但是，我们可以把接口当做是特殊的抽象类。而一个类实现一个接口，可以当做是特殊的继承。因此，从这个意义上来说，SpiderMan 这个类实现 Spider 接口，就是一种特殊的继承。因此，SpiderMan 继承了一个类 Man，又用一种特殊的方式继承了一个特殊的父类 Spider。这样，SpiderMan 就相当于继承自两个类。

上面的例子说明，由于实现一个接口，相当于继承自一个特殊的父类。因此，在 Java 语言中，我们可以使用接口，来实现了概念上的多继承。

既然 C++语言能够直接实现多继承，为什么 Java 语言要摒弃多继承这个特性，而要用接口这种语法来间接的实现多继承呢？

首先，使用接口实现多继承，能够区分主要类型和次要类型。考虑下面这个继承树：



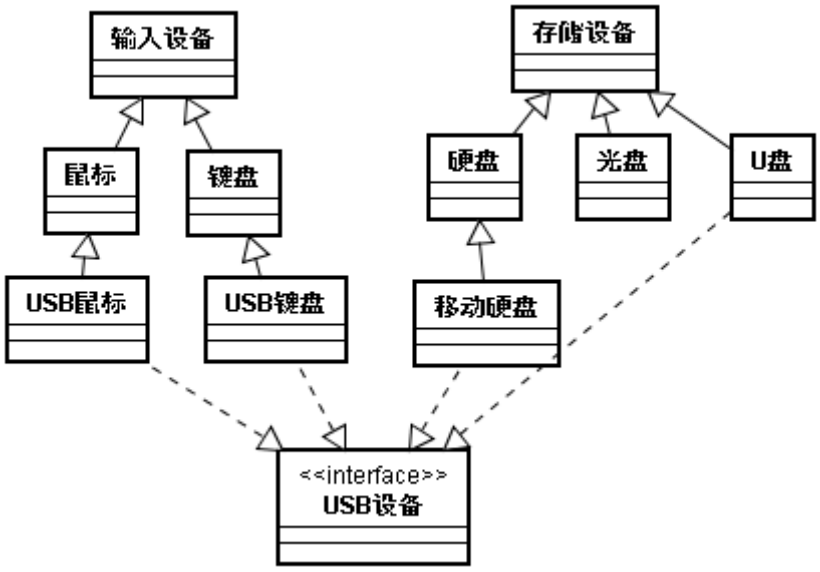
在上面这个继承树中，父类体现了设备的共性，而子类体现了特性。例如，USB 鼠标是特殊的鼠标，USB 键盘是特殊的键盘。而对键盘和鼠标这两个类提炼出共性，则得到父类“输入设备”。也就是说，键盘和鼠标都是特殊的输入设备。

再例如，移动硬盘是特殊的硬盘，而硬盘、光盘以及 U 盘，这些具体的子类抽象出共性，成为“存储设备”这个类。

我们可以看到，在上面的这几个类中，我们对一些具体的子类提炼出共性，从而形成了上面的继承树。

然而，这些类除了继承树中表现出的共性之外，还有其他的共性。

例如，USB 鼠标、USB 键盘、移动硬盘、U 盘，这些设备有一个共性：这些设备都能够通过 USB 端口与电脑相连，因此，他们除了各自的主要作用之外，有个额外的共性：他们都是 USB 设备。为了表示这个关系，我们设计一个接口：USB 设备，让上述四个设备实现这个接口。如下图：



这样，就利用接口实现了特殊的多继承。但是，虽然我们可以把实现接口当做特殊的继承，但是事实上，实现接口与继承父类相比，是处于相对“次要”的地位。这样，就能够区分“主要类型”和“次要类型”。

怎么来理解“主要类型”和“次要类型”呢？例如，对于“移动硬盘”来说。我们购买移动硬盘的主要目的，是为了存储数据。因此，“存储设备”是其主要类型。当然，为了让设备与电脑主机之间能够更加方便的交互，让移动硬盘实现“USB 设备”这个接口也是非常必要的。然而，相对于存储数据，用 USB 连接是一个次要的功能，因此，相对于“存储设

备”，“USB 设备”这是一个次要类型。我们把“USB 设备”定义为一个接口，就可以区分主要类型和次要类型。

在 Java 中，能够很容易的区分一个类的主要类型和次要类型。我们可以让一个类继承自其主要类型，而次要类型，可以作为接口，让这个类来实现。

例如，对于 SpiderMan 这个类来说，产生这个类的原因是一个 Man 类的对象受到了一些外界的影响（蜘蛛侠具有蜘蛛能力的原因，是因为小时候被蜘蛛咬了一口……），产生了变化，从而实现了 Spider 接口。在这个过程中，Man 是主要类型，而 Spider 是次要类型，是接口。

相对应的，对于“忍者神龟”这个类来说，产生这个类的原因是，四个乌龟对象受到影响之后，实现了“忍者”这个接口（四只动物园的海龟，被化学药品影响而成为了“忍者神龟”）。因此，对于忍者神龟来说，乌龟是主要类型，其次实现了“忍者”接口。

另外，通过“存储设备/输入设备”的例子，我们还可以看出，从概念上说，接口是怎么设计出来的。首先，我们在介绍继承关系的时候曾经说过，“父类”，从设计上说，是对多个不同子类的共性的抽象。在上面这个例子中，我们对“硬盘”、“光盘”、“U 盘”等子类进行了共性的抽象，抽象出“存储设备”这个父类来。然而，我们还可以对“移动硬盘”、“U 盘”、“USB 鼠标”等设备再一次进行共性的抽象，从而抽象出“USB 设备”这个接口。也就是说，父类可以认为是对子类主要共性的抽象，而接口可以认为是对子类次要共性的“再抽象”。

另外，单继承相对多继承的好处，就在于单继承具有简单性。使用单继承，类与类之间能够形成简单的树状结构。而对于多继承，类和类之间的关系相对要复杂的多，很有可能会形成复杂的网状结构。但是，用接口实现的多继承，则不会破坏类之间树状结构的简单性。这是因为这棵树是由类之间形成的，是事物主要类型所组成的关系。一个类实现再多的接口，有再多的次要类型，也不会改变其主要类型之间的树状结构。

例如，在生活中，每个家族的家谱都能够形成简单的树状结构，原因在于，在记录家谱的时候，只考虑一个人的亲生父亲。当然，人也可以认干爹。但是，干爹再多，也不会写到家谱中。因为干爹毕竟不是亲爹，在记录家谱的时候，亲爹是主要类型，而干爹只能是次要的，忽略不计的。

因此，在 Java 中，可以通过实现接口的方式来实现多继承。这种语法设计相对于 C++ 来说先进的多，因为，用接口实现多继承不会破坏类之间树状关系的简单性。

2.2 接口与解耦合

除了实现多继承之外，接口最重要的作用就是解耦合。什么叫解耦合呢？我们看下面这段代码的例子。

定义若干灯泡类。代码如下：

```
class RedBulb {
    public void shine(){
        System.out.println(“Shine in Red”);
    }
}
class YellowBulb{
    public void shine(){
        System.out.println(“Shine in Yellow”);
    }
}
```

```
class GreenBulb{
    public void shine(){
        System.out.println("Shine in Green");
    }
}
```

然后创建一个台灯类，首先装上红灯泡，代码如下：

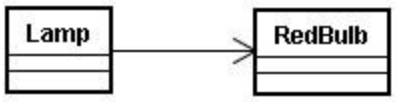
```
class Lamp{
    private RedBulb bulb;
    public void setBulb(RedBulb bulb){
        this.bulb = bulb;
    }
    public void on(){
        bulb.shine();
    }
}
public class TestLamp{
    public static void main(String args[]){
        Lamp lamp = new Lamp();
        RedBulb rb = new RedBulb();
        lamp.setBulb(rb);
        lamp.on();
    }
}
```

在上面的代码中，lamp 的 on 方法，调用了 bulb 的 shine 方法。也就是说，当我们调用台灯对象的“开”方法时，台灯对象会去调用灯泡对象的发光方法。

但是，问题来了：现在的这个台灯，装的是红灯泡。如果现在想要把红灯泡换成绿灯泡，应当如何操作呢？

由于 Lamp 类中的 bulb 属性是 RedBulb 类型，因此，一旦要修改，则必须要把 Lamp 类中的属性类型进行修改，并且修改 setBulb 方法的相应参数和实现。也就是说，当我们希望把 bulb 属性由 RedBulb 替换为 GreenBulb 的时候，必须修改 Lamp 类的代码。

也就是说，如果要想更换不同种类的灯泡，就要修改台灯的内部结构！这无疑是跟现实生活不相符合的。之所以产生这样的矛盾，原因在于 Lamp 类与 RedBulb 类型紧密联系在一起，形成了强耦合的关系，如下图：



下面我们使用接口来解决这样的问题。首先，定义 Bulb 接口：

```
interface Bulb{
    void shine();
}
```

然后，修改三种灯泡的代码，让他们都实现 Bulb 接口。

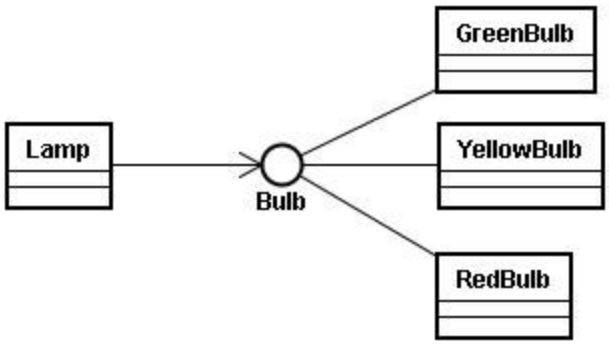
```
class RedBulb implements Bulb{
```

```
    public void shine(){
        System.out.println("Shine in Red");
    }
}
class YellowBulb implements Bulb{
    public void shine(){
        System.out.println("Shine in Yellow");
    }
}
class GreenBulb implements Bulb{
    public void shine(){
        System.out.println("Shine in Green");
    }
}
```

之后，修改 Lamp 类，并给出 TestLamp 类的代码。

```
class Lamp{
    private Bulb bulb;
    public void setBulb(Bulb bulb){
        this.bulb = bulb;
    }
    public void on(){
        bulb.shine();
    }
}
public class TestLamp{
    public static void main(String args[]){
        Lamp lamp = new Lamp();
        Bulb b1 = new RedBulb();
        lamp.setBulb(b1);
        lamp.on();
        Bulb b2 = new GreenBulb();
        lamp.setBulb(b2);
        lamp.on();
    }
}
```

Lamp 类的 bulb 属性被改为接口类型 Bulb，从而，Lamp 类与具体的实现类之间用 Bulb 接口分离开了。当 Lamp 类希望把 bulb 属性由 RedBulb 对象变更为 GreenBulb 对象时，不需要修改任何自身代码。只需要调用 setBulb 方法，接受不同的 Bulb 接口的实现类就可以了，从而，Lamp 类通过 Bulb 接口，实现了与 Bulb 实现类的弱耦合。如下图所示：



这样，我们就利用接口，把原来强耦合的关系，变为了弱耦合的关系。这就是接口最重要的作用：解耦合。

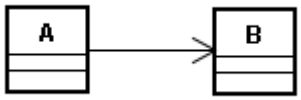
由于接口中所有的方法都是抽象方法，因此，定义一个接口，可以看作是定义了一个标准。它只定义了，一个对象应该具有哪些方法，而丝毫没有定义对象如何实现这些方法。方法的实现统统交给接口的实现类来完成。这样，接口的出现，就阻隔了接口使用者和接口实现者之间的耦合关系。当接口实现者变化的时候，对接口使用者不产生任何影响。

在生活中，对象之间的弱耦合关系也是通过标准来实现的。例如，当电脑的硬盘出现故障的时候，我们可以很容易的为电脑更换一块其他品牌的新硬盘，而对电脑的主板 CPU 等元件不产生丝毫影响。这显然是因为，不同的硬盘厂商在生产自己的硬盘产品的时候，都会遵循统一的标准，如电气接口的规格，硬盘产品的尺寸等等。试想一下，如果没有了这些标准，各个厂商各自为战，生产出规格各异的硬盘产品，那么我们在更换硬盘的时候，是不是就没有了那么多的选择了呢？

3 接口回调

有了接口和多态之后，对我们的开发模式和开发思路都有着很深刻的影响。在企业级开发应用中，肯定不会把所有的功能都写在一个模块里，也肯定不会把所有的功能都让一个程序员来完成。因此，就有了程序员之间的分工和合作。

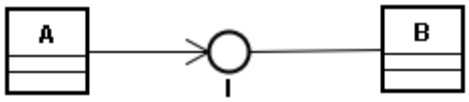
假设，现在有两个类需要完成，这两个类一个是 A 类，一个是 B 类，而 A 类需要调用 B 类提供的方法。示意图如下：



而现在有两个程序员，一个张三，一个李四，在项目经理的分配之下，两个人分别负责者两个类的编码。其中，张三负责 A 类，李四负责 B 类。

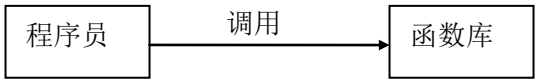
由于 A 类要调用 B 类提供的功能，因此，在李四没有完成 B 类的代码之前，A 类根本无法使用 B 类，必须要等到李四开发 B 类完成，张三才能够开始开发 A 类。也就是说，A 类作为功能的使用者，必须等到功能的实现者 B 类完成之后，才能进行开发。

但是，有了接口之后，我们可以用一种新的开发方式来解决这个问题。我们可以在开发之前，先设计一个接口 I，定义 B 类中应该具有哪些方法，让 B 类来实现接口 I。而利用 I，把 A 类和 B 类之间的紧耦合关系转为弱耦合关系，示意图如下：



有了 I 接口之后，张三在开发 A 类的时候，并不需要等待李四的 B 类开发完成。只要在 A 类中使用 I 接口类型，等 B 类开发完成之后，可以用多态调用 B 类中的实现。在这种情况下，A 类作为 I 接口功能的使用者，而 B 类作为 I 接口功能的实现者，在开发过程中，并不需要强调谁先谁后。A 类的开发者，完全可以在 B 类没有完成的情况下，就开始使用 B 类的方法（因为这些方法已经在 I 接口中定义了）。甚至于，有可能在没有开发 B 类的情况下，A 类就已经完成开发了。

这个改变，对程序开发来说，有着非常重大的影响。例如，在介绍“函数”时，我们提到，函数是面向过程中一个很重要部分。对于一个成熟的面向过程的语言来说，应当提供大量的函数库，让程序员使用。事实上，面向过程的程序员，都是去调用函数库中的函数，来完成自己所需要的功能。示意图如下：



也就是说，程序员在开发过程中，一直是扮演着“调用者”的角色。原因也很简单，面向过程的编程方式中，必须先把功能实现了，然后程序员才能去使用功能。

那能不能反过来呢？让程序员提供一些代码，而让系统提供的函数或者类，来调用程序员写的代码？也就是说，让程序员成为功能的提供者，而让系统成为功能的使用者？

上面的想法有什么意义呢？我们看下面这个例子：

例如，在 Java 中，可以利用 java.util.Arrays.sort 方法，来对数组进行排序。我们在数组部分的学习中，曾经为大家介绍过冒泡排序算法。冒泡排序，是所有的排序方法中最简单的排序方法，也是执行效率最低的方法。在计算机科学领域，有很多相对已经比较成熟的排序算法。这些算法都比冒泡排序要高效的多，但是也要比冒泡排序算法复杂的多。幸运的是，Sun 公司提供了一个 java.util.Arrays.sort 函数，这个函数能够对数组进行排序，排序时使用的算法，是一种经过调优的快速排序算法。这个函数的使用如下：

```
public class TestArraySort{
    public static void main(String args[]){
        int[] a = {1，7，2，5，3};
        printArray(a); //输出 1  7  2  5  3

        java.util.Arrays.sort(a);
        printArray(a); //输出 1  2  3  5  7
    }

    public static void printArray(int[] a){
        for(int i = 0 ; i < a.length ; i++){
```

```
            System.out.print( a[i] +"  ");
        }
        System.out.println();
    }
}
```

我们可以看到，这个 java.util.Arrays.sort()能够对 int 类型的数组进行排序。同样的，这个函数也能够对其他的一些基本类型的数组（例如 byte[]，double[]等）进行排序。

那么除了基本类型之外，Arrays.sort()能不能对对象类型的数组进行排序呢？

首先，如果要进行排序的话，除了排序算法之外，有一个最基本的要素：排序规则。简单的来说，因为排序是把一个数组中的元素从小到大依次排列，因此，必须要有一个途径，能够比较两个元素的大小。只有区分出元素的大小之后，才能够把小的元素放到前面，大的元素放在后面，从而最终让数组元素从小到大排列。

两个元素比较大小的方式，叫做排序规则。对于基本类型来说，排序规则就是数学上的比较方式。例如，如果要让 int 变量 a 和 int 变量 b 进行比较，只要直接使用表达式(a>b);如果这个表达式为 true，则 a 比 b 大；如果这个表达式为 false，则说明 a 小于等于 b。

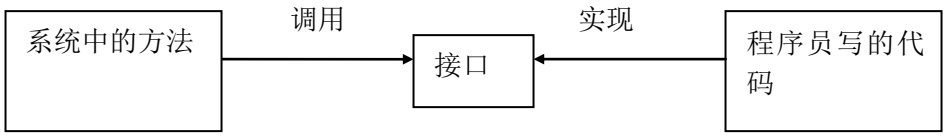
但是比较对象时，就没有那么简单。首先，两个对象之间，不能直接使用>，<，>=，<=等布尔运算符来比较大小。因此，程序员必须要自己指定两个对象如何比较大小。例如，当程序员创建了一个 Student 类之后，必须由程序员自己来指定排序规则，说明两个 Student 对象如何比较大小。假设有个两个 Student 对象，一个对象代表 18 岁的李四，另一个对象代表 20 岁的张三，这两个对象谁大谁小？程序员必须自己指定规则。

我们简单回顾一下刚刚说的内容。java.util.Arrays.sort 方法是一个高效的排序算法。但是，如果要对对象类型进行排序的话，则在 sort 方法中，必然会用到对象类型的排序规则。而这个规则，不是由 Sun 公司来定，是由程序员来定的。我们可以理解为，程序员提供排序规则，而在 Sun 公司提供的 sort 方法中，使用了这个排序规则。示意图如下：

可以看到，这跟传统的面向过程的模式有着很大的区别。在传统的开发模式中，系统提供函数库，让程序员调用；而在这种开发模式中，由程序员提供功能，让系统中的某些函数调用。

在 Java 中，我们可以利用接口实现这一点。

系统中事先定义好了一个接口，然后，在系统的方法中，调用这个接口中的方法。这样，系统中的方法就作为这个接口的使用者。而程序员实现这个接口，程序员就是接口的实现者。像这样，由程序员实现接口，由系统其他的类来通过多态进行调用，这种编程的方式，叫做接口回调。示意图如下：



例如，为了定义排序规则，Sun 公司定义了一个接口：java.lang.Comparable。这个接口用来表示一个对象的排序规则。我们编写的类应该实现这个接口。

Comparable 接口中只有一个方法：compareTo 方法，实现这个方法，就能规定两个对象如何比较大小。假设程序员要创建一个 Student 类，希望用 java.util.Arrays.sort()对一些 Student 对象进行排序，则要求 Student 类实现 Comparable 接口。示意图如下：

那么如何实现 Comparable 接口呢？我们给出 Student 类的示例代码：

```
class Student implements Comparable<Student>{
    int age;
    String name;
    public int compareTo(Student stu){
        //…
    }
}
```

这段代码有三个要注意的地方。

第一，在 Student 类实现 Comparable 接口时，后面有一个尾巴：“<Student>”。这部分是 Java5.0 提供的新特性，称之为“泛型”。这个特性在后面的还有详细的论述，此处不做过多的解释。

第二，在 Student 类中，必须要实现 compareTo 方法。这个方法接受一个 Student 对象作为参数。compareTo 方法用来比较两个对象，这两个对象一个是“当前对象”，另一个则是 compareTo 方法的参数。例如，假设有两个 Student 对象 stu1 和 stu2，则如果调用 stu1.compareTo(stu2)，就表示把 stu1 和 stu2 进行比较。其中“当前对象”就是指的 stu1 对象；而把 stu2 作为 compareTo 方法的参数，因此“参数”指的就是 stu2。

第三．compareTo 方法返回一个整数。这个整数的数值就表示比较的结果：如果返回值小于 0，则表明当前对象比参数对象小；如果返回值等于 0，则说明两个对象一样大；如果返回值大于 0，则说明当前对象比参数对象大。

例如，我们规定，对学生的年龄进行排序，年龄较小的学生排前面，年龄较大的学生排后面，则可以实现 compareTo 方法如下：

```
public int compareTo(Student stu){
    if (this.age > stu.age){
        return 1;
    }else if (this.age < stu.age){
        return -1;
    }else {
        return 0;
    }
}
```

定义完 compareTo 方法之后，就可以使用 Arrays.sort 方法进行排序了。例如下面的例子：

```
public class TestSort{
```

```
    public static void main(String args[]){
        Student[] ss = new Student[3];
        ss[0] = new Student(“Tom”, 18);
        ss[1] = new Student(“Jim”, 17);
        ss[2] = new Student(“Jerry”, 20);
        java.util.Arrays.sort(ss);
        for(int i = 0; i<ss.length; i++){
            System.out.println(ss[i].name + “ ” + ss[i].age);
        }
    }
}
```

在调用 Arrays.sort 方法时，根据我们规定的排序规则，把年龄小的学生排在前面，把年龄大的学生排在后面。

完整代码如下：

```
class Student implements Comparable<Student> {
    int age;
    String name;

    public Student(String name, int age) {
        this.name = name;
        this.age = age;
    }

    public int compareTo(Student stu) {
        if (this.age > stu.age) {
            return 1;
        } else if (this.age < stu.age) {
            return -1;
        } else {
            return 0;
        }
    }

}

public class TestSort {
    public static void main(String args[]) {
        Student[] ss = new Student[3];
        ss[0] = new Student("Tom", 18);
        ss[1] = new Student("Jim", 17);
        ss[2] = new Student("Jerry", 20);
        java.util.Arrays.sort(ss);
        for (int i = 0; i < ss.length; i++) {
            System.out.println(ss[i].name + " " + ss[i].age);
```

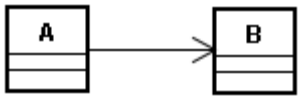
```
        }
    }
}
```

输出结果如下：

```
Jim 17
Tom 18
Jerry 20
```

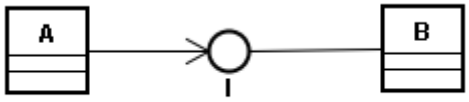
可以看到，输出的结果，对学生对象的年龄进行了排序。在这个程序中，很显然先有 Sun 公司为我们提供的 Arrays.sort 方法，然后才有 Student 类作为接口的实现者。这是一个非常典型的接口回调：程序员提供 Comparable 接口的实现，供 JDK 中 Arrays.sort 方法来调用。

简单的说，我们习惯于这样的编程方式：



由系统为我们提供 B 类，而我们负责编写 A 类来使用 B 类。

而接口回调为我们提供了新的方式：



由系统为我们提供 A 类和 I 接口，我们负责编写 B 类来实现 I 接口。A 类通过对 I 接口中方法的调用，利用多态，来调用我们所写的 B 类的方法。这不能不说是编程方式的一次伟大的突破。