JAVA 语言的多态性及其应用研究

An Study on Java's Polymorphism

蓝雯飞 周 俊 陈淑清 （中南民族大学计算机科学学院 430074）

摘要：多态性增强了程序设计语言的表达能力。本文就 JAVA 语言支持的多态性作了全面的阐述和讨论，还重点分析了抽象类和接口的区别；在此基础上，结合例子说明了多态性在程序设计中的应用。

关键词：Java 语言 多态性 重写 类

1 引言

在 Java 面世短短的十来年间，Java 以其简单、易用、面向对象、面向网络、健壮及可移植性强等诸多优点，被越来越多地推广和使用。

和 Java 语言的封装性和继承性相比，多态性显得更加重要，而且不易掌握。本文着重就 J ava 语言的多态性及在程序设计中的应用作一个深入的探讨。

多态性（polymorphism）一词来源于拉丁语 poly（表示多的意思）和 morphos（意为形态），其字面的含义是多种形态。从广义上来说，多态性是指一个名字可以表示不同但相似的语义。它反映了人们在求解问题时，对相似性问题的一种求解方法。在面向对象中它是指不同但相似的对象收到同一消息将导致完全不同的行为[1]。

2 Java 语言的多态形式

在 Java 语言中，多态性包括重写多态、重载多态、强制多态 3 种形式。

重写多态称为通用多态性，通用多态性用来系统地刻画语义上相关的一组类型。重载多态和强制多态统称专用多态性，专用多态性刻画语义上无关联的类型间的关系。

重写多态在 Java 语言中是从（公有）继承派生出的概念，和 C + + 语言不一样，Java 语言只支持公有继承。一个类可以直接继承一个超类或多个接口，通过继承得到的类称为子类。如：

class D extends P {……}

表示类 D 是超类 P1 的子类。通过继承，超类的操作适用于子类，也就是说子类的对象可出现在超类对象可以出现的一切场合，这就是所谓的多种形态（多态）。

重载是指用同一个名字命名不同的方法，但每个方法的参数表应该不同，例如：

```
class A
{ public int max(int x,int y){.....} //求两整数的最大数
  public int max(int x,int y,int z) {.....} //求三整数的最大数
  public double max (double x, double y) {.....} //求二实数的最大数
}
```

从某种意义上说，重载只是一种出于方便的语法上省略。

强制是指将一种类型的值转换成另一种类型的值进行的语义操作，从而防止类型错误。

类型转换可以是隐式的，如语句"D = I"是把整型变量 I 的值转换为实型值来更新实型变量 D；也可以是显式的，如类型强制表达式"(int)(x + y)"强制编译器将表达式 x + y 的值强制为整型值。

从总体上来说，通用多态性是真正的多态性；专用多态性只是表面的多态性。因为重载只允许某一个符号有多种类型，而它所代表的值分别具有不同的、不相兼容的类型。类似地，类型转换也不是真正的多态，因为在操作开始前，各值必须转换为要求的类型，而输出类型也与输入类型无关。相比之下，公有继承却是真正的多态。

3 Java 语言的多态性应用

3.1 重写多态

在 Java 语言中，重写多态是从类的(公有)继承导出的概念，意指类 S(公有)继承类 T，那么我们便可写出一个程序段，它既能够处理类型 T 的对象，也能够处理类型 T 的子类 S 的对象，该程序段我们就称之为多态程序段。

在继承方式下，子类继承超类的所有操作，或者说，超类的操作能被用于操作子类的对象，然而，在大多数的情况下，子类中往往还声明有新的字段，超类的方法就不能适应子类的需要，此时，子类需重写超类的方法，见下例中的 void rectangle::showarea()。

```
class Shape
{ private double x,y;
  public Shape(double x1,double y1) { x=x1;y=y1;}
  public void showarea() {"Area of Shape is:" + 0.0);}
}
class Rectangle extends shape
{ private double w,h;
  public Rectangle(double x,double y,double w1, double h1)
  { super(x,y);w=w1;h=h1;}
  public void showarea()
  { System.out.println("Area of rectangle is:" + w*h);}
}
public class Test
{ public static void disparea(Shape r) { r.showarea();} //多态程序段
  public static void main(String args[])
  { Rectangle r=new Rectangle(1,1,2,3); disparea(r);}
}
```

在 main()方法体内，用子类对象的引用 r 作为实际参数调用方法 disparea()，从而能求得矩形的面积并显示在屏幕上，象 disparea()这样的方法就是多态程序段，在 Java 程序设计中经常出现。

利用 Java 语言的这种多态性，可方便地扩充程序，这也是 Java 语言的主要特点之一。例如，如果还需要求形状圆的面积并显示在屏幕上，我们可这样扩充上述程序，从超类 Shape 派生圆类 Circle，并在圆类中重写求并显示面积的函数 showarea()，那么经扩充后的程序就可方便求得圆的面积。整个程序由于篇幅有限，就不给出了，请读者自己考虑。

从上面的的讨论中我们可看出，重写(Overriding)是超类与子类之间多态性的一种表现，如果在子类中定义某方法与其超类有相同的名称和参数，我们说该方法被重写(Overriding)。若超类的对象调用这个方法时，则调用子类中的定义；若子类的对象使用这个方法时，则调用子类中的定义，对它而言，超类中的定义如同被"屏蔽"了。正是这个多态特点，使我们可为同一继承结构中的不同类，编写一个多态程序段，来统一处理不同类的对象，从而能达到对不同的对象，使用相同的处理逻辑，这是非面向对象程序设计语言不具备的。

可见，多态性为我们统一地处理(如 disparea()方法)这组相似的 showarea()方法提供了一种技术支持，使我们的设计逻辑简单明了、可读性强。

这里关键要理解好最顶层类的设计和作用。它不仅通过继承性为下层的类提供了代码共享的手段、减少了重复性开发、提高了软件的开发效率。另一方面，通过多态性提供了统一地处理该类层次中同名函数的机制，简化了处理这些同名函数的处理逻辑，同样也提高了软件的开发效率。正是因为这一点，即为了能统一地处理该类层次中同名的方法，在 Java 语言中，增加了如下两种语法机制以便更好地设计多态性：

3.1.1 抽象方法

在很多应用中，类层次的顶层类并不具备下层类的一些功能。我们可在超类中将这些方法声明为没有实现的抽象方法，含有抽象方法的类就叫抽象类。这样，就可通过顶层类提供统一处理该类层次的方法。例如，在上面的例子中，更合理的设计就是将 Shape 类中的 showarea()方法定义为抽象方法，因为 Shape 类它不代表某些具体的对象。

```
abstract class Shape //抽象类
{ private double x,y;
  public Shape(double x1,double y1) { x=x1;y=y1;}
```

```
    abstract public void showarea( ) ;
  }
  class Rectangle extends shape
  { private double w,h;
    public Rectangle( double x,double y,double w1,
double h1)
    { super(x,y) ;w = w1;h = h1;}
    public void showarea( )
    { System. out. println( "Area of rectangle is:" +
w * h) ;}
  }
  public class Test
  { public static void disparea ( Shape r ) { r.
showarea( ) ;} //多态程序段
    public static void main( String args[ ])
    { Rectangle r = new Rectangle(1,1,2,3) ; dis-
parea(r) ;}
  }
```

这样,我们就可按如下的方式统一地处理该类层次的计算并显示面积的功能:

```
void disparea( Shape r) { r. showarea( ) ;}
```

一般地,在顶层的超类中,总有很多的这种抽象方法,它为其他子孙类用抽象机制实现多态性提供了一个统一的界面。这是设计多态性经常用到的模式。

由于抽象类(如 Shape)不能产生对象,因此方法 disparea( ) 用来统一操作派生类 Rectangle 和 Circle 的对象。类 Shape 的设计尽管是用继承性语法表达的,但它的主要目的不是为代码共享而设计的,而是为使用多态性而设计的,它是另一个维度的抽象。

3.1.2 接口

接口是 Java 中支持多态性的另一重要概念。接口允许多继承,也就是说,当类或接口用关键字 implements 或 extends 从接口继承时,它可以同时有多个父接口,这一点与类继承是不同的。通过接口继承我们可以实现接口的组合与扩充。接口常常被用来为具有相似功能的一组类,对外提供一致的服务接口,这一组类可以是相关的,也可以是不相关的,而抽象类则是为一组相关的类提供一致的服务接口。所以,接口往往比抽象类具有更大的灵活性。

我们在使用抽象类和接口时,必须注意以下 3 个方面:

(1) 抽象类中可以包含方法的声明,也可以提供方法的实现代码,而接口中只能提供方法声明,不可以有任何实现代码;

(2) 抽象类与其子类之间存在层次关系,而接口与实现它的类之间则不存在任何层次关系;

(3) 抽象类只能被单继承,而接口可以被多继承。

Java 的接口既和继承有关又和多态相关,而且是众多的面向对象程序设计语言不支持的,是 Java 的一个重要特征,因此接口是运用 Java 的难中之难。

在上面的例子中,将 Shape 定义为抽象类是恰当的。但当作为一个类型声明的时候,接口拥有抽象类无法与之较量的灵活性。我们来看一个具体例子,下面是接口 Action 的声明:

```
interface Action
{void close( ) ;
 void open( ) ;
}
class Computer implements Action
{public void close ( ) { System. out. println
("Computer close!") ;}
  public void opent ( ) { System. out. println
("Computer open!") ;}
}
class Window implements Action
{public void close( ) {System. out. println( "Win-
dow close!") ;}
  public void open( ) {System. out. println( "Win-
dow Open!") ;}
}
public class Test
{ public static void close( Action r) { r. close( ) ;}
//多态程序段
  public static void main( String args[ ])
  { Computer r1 = new Computer( ) ; close( r1) ;
   Window r2 = new Window( ) ; close( r2) ;}
}
```

从上述代码可以看出,接口可以派生不相关的两个类,即窗口类 Window 和计算机类 Computer,但它们实现相同行为 open 或 close。因此接口可以实现更加灵活多样的多态性。接口还可以避免盲目类继承所带来的潜在危险。比如,当我们不了解一个类的全部属

性和方法，对类进行盲目扩充时，同样可能产生冗余代码，而且在继承我们所需的属性和方法的同时，有可能意外继承一些不需要的方法和属性，这在编译中不会报错，但方法被类对象调用时会产生一些不可控制的后果。而接口可以避免这些问题的产生，它要求实现接口的所有类必须实现接口中的所有方法，否则该类不可以被实例化，该类必须以 abstract 关键字声明，不然编译时报错。可以说接口是接口用户（使用接口的程序）和接口实现者（实现接口的类）之间一致的约定，比抽象类更加安全，清晰，不存在盲目性[2]。就这些原因来说，当不需要为一组类提供公用实现代码时，我们优先考虑接口，以使用 Java 语言的多态性。

抽象类和接口是 Java 语言中的二个重要的对象引用类型，是 Java 程序设计使用多态性的基础。

3.2 重载多态

一般的面向对象语言都提供重载多态，重载是多态性的最简形式，它把更大的灵活性和扩展性添加到程序设计语言中，和 C++ 语言不太一样，Java 只提供方法重载。

重定义已有的方法称方法重载。如对构造方法进行重载定义，可使程序有几种不同的途径对类对象进行初始化；下面的例子给出了点类 Point 对构造方法的重载。

```
class Point
{ int x,y;
  public Point( ){x=0;y=0;}// 重载构造方法 1
  public Point(int a,int b){x=a;y=b;} // 重载构造方法 2
  public Point(Point r){x=r.x;y=r.y;} // 重载构造方法 3
  ......
}
```

对于上面定义的 Point 类，我们可根据需要调用不同的构造方法，下面的三个语句将分别调用不同的重载构造方法初始化类对象。

```
Point p1=new Point( ); //调用无参构造方法
Point p2=new Point(3,4); //调用一般构造方法
Point p3=new Point(p1); //调用拷贝构造方法
```

3.3 强制多态

强制也称类型转换。Java 语言定义了基本数据类型之间的（隐式/自动）转换规则。隐式转换规则为：

byte -> short -> int -> long -> float -> double
↑
char

上述转换原则表明，只允许最大值较小的数据类型向最大值较大的数据类型转换。

当在程序中要按照程序员的意愿进行类型转换时，则必须使用强制（显示）类型转换表达式"(T)E"来改变编译器所使用的隐式规则，以便按自己的意愿进行所需的类型强制。其中 E 代表一个运算表达式，T 代表一个类型表达式。例如，设变量 f 的类型为 double，且其值为 3.14。则表达式(int)f 的值为 3，类型为 int。

在进行类型转换时，需要注意的是，Java 允许除布尔型以外的任何基本数据类型强制转换成其他任何一种基本数据类型[3]。此外，强制会使类型检查复杂化，尤其在允许重载的情况下，可能会导致无法消解的二义性，在程序设计时要注意避免由于强制带来的二义性[4]。

4 结束语

多态性是 JAVA 语言中最重要、也是最难掌握的特性。在 JAVA 面向对象的程序设计中有效地利用多态性，可以提高程序的可扩充性、灵活性和程序代码的再用率。而一般地，JAVA 语言书中只将继承所支持的多态性称为多态性，尽管也说明了重载和类型强制语法，但没有说明它们也是多态性的一种，这样 JAVA 的程序设计人员就不能以多态性为指导开发软件，不能充分地发挥其优势。本文就 JAVA 语言支持的多态性作了统一的描述和讨论，还重点分析了抽象类和接口的区别。希望 JAVA 的程序设计人员能更好地利用多态性，设计、开发出更好的 JAVA 程序。

参考文献

1 蓝雯飞，C++ 面向对象程序设计中的多态性研究[J]，计算机工程与应用，2000(8)：97。

2 陈淑清、蓝雯飞，深入剖析 Java 语言的抽象类与接口[J]，微计算机应用，2004(5)。

3 刘正林，Java 技术基础[M]，华中科技大学出版社，2002：75。

4 蓝雯飞，面向对象程序设计语言 C++ 中的多态性[J]，微型机与应用，2000(6)：11。