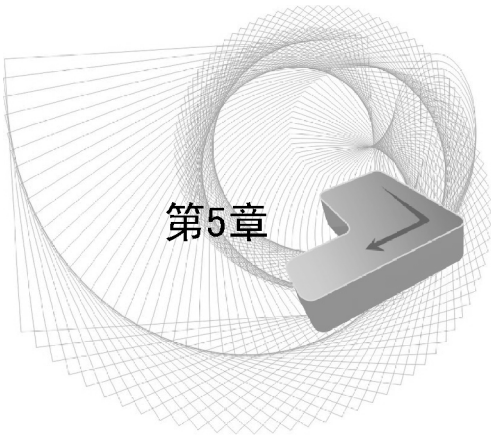

第5章

chapter 5

继承和多态

本章重点内容导读

- ◇ 继承、重写的实现方法。
- ◇ 向上和向下转型以及面向父类编程的设计方法。
- ◇ 子类的初始化顺序。
- ◇ 多态的实现方法及应用。
- ◇ 抽象类型与接口的区别。
- ◇ 内部类和匿名内部类的适用性和应用方法。
- ◇ 异常的分类和具体应用场景。
- ◇ 泛型的应用场景和接口替换方法。

在了解了什么是类、如何定义以及了解封装概念之后，下面将接触更有趣的面向对象核心也是难点：继承和多态。

面向对象的三个核心：封装、继承和多态。

5.1 继承的重要性

继承是为了重用代码，继承不是重用代码的唯一方式。

重用(复用)是面向对象语言所宣称的技术核心之一。通过继承可以重用代码，可以减少重复代码的出现，维护更加方便。重用代码可以共享相同一段代码，如果更改此处代码，所有使用该段代码的客户将会自动更改，这在大规模系统的代码维护中带来了极大的方便。

重用代码不是复制代码。复制只是简单地拷贝、粘贴，当复制的越来越多，更改起来将会非常困难，因为重复代码遍布整个工程，算法或者功能块一旦有问题，将修正非常多的地方，极易出错。

如图 5-1 所示，通过继承，子类将会自动拥有父类的服务，更改父类的服务，子类会自动更新，子类可以扩展自己的服务来满足新的需要。

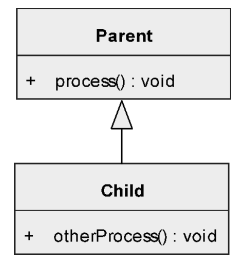

图 5-1 继承重用父类服务

> 重用有组合和继承两种技术手段。

5.2 组合重用

组合：顾名思意，将多个需要重用的事物放在一起，联合彼此的功能完成整个任务。实现代码也非常简单。如图 5-2 所示，通过组合关联关系将各部分零件组合在一起，形成一个完整的车，相当于车这个对象重用了零件的功能。

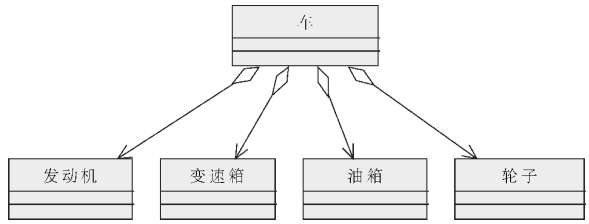

图 5-2 通过组合重用类的功能

例 5-1 通过组合合成一个汽车。

```
//Car.java
package hust;

public class Car {
    private Wheel wheel;
    private Engine engine;
    private Transmission transmission;
    private Tank tank;

    public Car(Wheel wheel,Engine engine,Transmission transmission,Tank tank) {
        this.wheel = wheel;
        this.engine = engine;
        this.transmission = transmission;
        this.tank = tank;
    }

    public void travel() {
        //调用各种设备功能使车辆行驶
    }
}

//Wheel.java
package hust;

public class Wheel {
}

//Engine.java
package hust;

public class Engine {
}

//Transmission.java
```

```
package hust;

public class Transmission {
}

//Tank.java
package hust;

public class Tank {
}
```

本例中，车辆的所有功能都是设备零件聚合而成的。

聚合关联在代码实现上就是将所有的零件类都作为汽车的成员变量，汽车通过重用所有零件的功能，完成了一个汽车自身的功能。

组合关联重用可以细分为两种：组合(Composite)和聚合(Aggregate)。

(1) 聚合 UML 符号：  。

(2) 组合 UML 符号：  。

从汽车这个例子可以看出，汽车实际上并不存在，它都是由各种已经存在的零件聚合而成的。零件装配在一起，汽车就存在，将零件拆解，零件依然存在，但汽车就消失了。这种关系称为聚合。

> 聚合中，所有成员实例的存在不依赖宿主类，成员类实例可以单独存在，而宿主类的存在依赖于成员类实例。

例 5-2　车辆的实例化。

```
public static void main(String[] args) {
    Wheel wheel = new Wheel();
    Engine engine = new Engine();
    Transmission transmission = new Transmission();
    Tank tank = new Tank();
    Car car = new Car(wheel,engine,transmission,tank);
    car.travel();
}
```

从实例化 Car 类型的代码中可以看出所有的零件都是在 Car 实例化之前完成的。也只有在这些零件都存在之后，才能进行 Car 的实例化工作。Car 即使没有创建或者消失，也不影响这些零件的存在。

另一个组合的例子如图 5-3 所示，通过聚合关联关系重用类的功能。

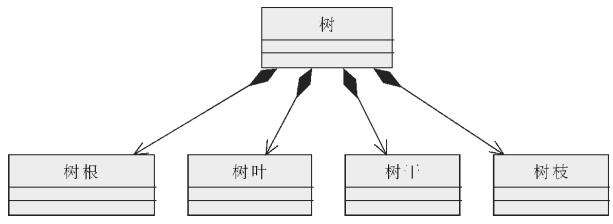

图 5-3　通过组合重用类的功能

例 5-3 通过组合形成的大树。

```
//Tree.java
package hust;

public class Tree {
    private Root root = new Root();
    private Leaf leaf = new Leaf();
    private Trunk trunk = new Trunk();
    private Branch branch = new Branch();

    public Tree() {
    }

    public void grow() {
        //root,leaf,trunk,branch 参与生长过程
    }

}

//Root.java
package hust;

public class Root {

}

//Leaf.java
package hust;

public class Leaf {
}

//Trunk.java
package hust;

public class Trunk {
}

//Branch.java
package hust;

public class Branch {
}
```

树是由树根、树叶、树干以及树枝组成的，这些成员的存在依赖树的存在，当树不存在了，这些成员也就不复存在了，因为它们没有单独存在的意义，这种关系称为组合。

> 组合中，所有成员实例的存在依赖宿主类，成员类实例不可以脱离宿主类单独存在，而宿主类存在时，成员类实例存在才有意义。

可以看出 Tree 中的成员都是在类内部实例化的，当 Tree 消失后，其中的所有成员实例也都失去了存在的意义。这是与上面聚合例子有显著的区别的。

例 5-4 Tree 的实例化。

```
public static void main(String[] args) {
    Tree tree = new Tree();
    tree.grow();
}
```

代码很简单，所有成员的实例化都在内部操作。

通过关联关系，使多个类共同协作完成一个复杂的任务，同样达到了重用代码的目的，而且看起来非常灵活。

> 另一种重要的重用方式就是继承。

复杂的协作关系都是通过关联产生的，而不是通过继承，继承只是为了创建一个新的相似类型，而不是一个协作。在可能的情况下，尽量使用组合方式重用代码，而不要使用继承的方式。组合方式可以更灵活地控制重用类，而继承方式可能相对更僵化些(注意，只是相对来说)。

组合和继承重用代码的区别非常明显。组合重用代码的方式非常灵活，类型彼此之间不需要非常强烈的依赖关系，在运行期间也可以灵活地替换，所以需要重用某一个类型的功能时，首先想到的应该是组合重用。继承是一种强烈的依赖关系，子类依赖父类，父类的更改影响子类，而且子类重用父类的部分代码也只能存在于子类中，没有太多的灵活性而言，除非继承能够带来明显的益处，否则推荐使用组合方式重用代码。

5.3 继承的定义

继承是面向对象语言中很重要的特性之一，它是多态性的基础。有了继承，才有了丰富多彩的设计方式。这种方式带来的是更大的扩展性、可维护性。

继承的概念同现实生活中的继承有些类似。

> 继承：是两个实体间的一种关系，其中一个实体是基于另一个实体而定义的。这种实现方式使父类的代码得以重用。

Java 中只能基于一个实体类继承，这点与 C++不同。

子类也称为父类的派生类，父类派生了子类。父类通常也称为基类。如图 5-4 所示，Shape 称为父类，其他的类称为 Shape 的子类。

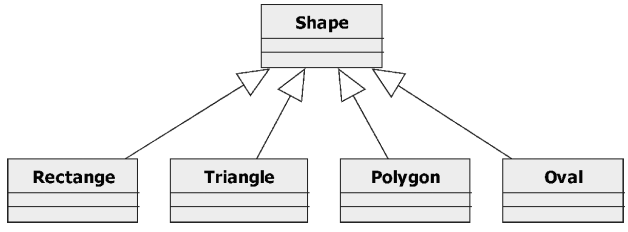

图 5-4 父类派生多个子类

基类的提取过程是一个将事物一般化的过程，因为所有的子类都将具有父类的特征，如果父类不具有一般性，那子类就没有理由去继承它，基类也就失去了存在的意义。

父类和子类的关系是一种一般化(泛化：Generalization)的关系。父类一般化(泛化)了子类，子类被父类一般化了。

例如，几何形状 Shape 类，在定义的时候，要将所有形状的一般化特征定义出来，而不能存在特别的特征(例如，椭圆形才具有的焦点特征)。也只有这样，所有的子类才可以被称为形状(Shape)。如果一个普通形状具有了一个椭圆形的焦点，那将是多么让人诧异的事情。

从泛化和 UML 图形来看，似乎父类是在多个子类先存在的情况下进行泛化，抽取一般性特征后才被定义出来的。确实，在开发过程中父类通常情况下并不是一开始就显而易见的，而是在不断的重构过程中，根据情况抽取出来的。一般性开发中，都是先存在普通的类型，例如 Rectangle、Triangle 等，这些类型出现多了，共性的特征多了之后，泛化子类为父类，才变得更有意义。

由于子类继承父类，也就具有了父类的共性特征，所以 Rectangle、Triangle 等也可以被称为 Shape。但是反之 Shape 不能称为 Rectangle。

子类可以默认是一个父类，但父类不能默认是一个子类。

```
Shape rect = new Rectangle();              //正确
Rectangle rect = new Shape();              //错误
```

如图 5-5 所示，从这些插座看，哪个更通用些，更具一般性呢？当然是第一个插座，但是它是否可以被定义为父类呢？当然不能。

> 一般化并不等同于多功能(通用)，将问题一般化的过程是将事物的特征和行为一般化的过程，而不是将所有特征集于一身的过程。

这个插座的父类应该是一个描述：是指有一个或一个以上的电路接线可插入的座，通过它可插入各种接线，便于与其他电路接通。它是一个抽象的类型，而不是一个具体的类型，通过继承才可以成为一个具体的、实用的插座。

继承的 UML 表示如图 5-6 所示。

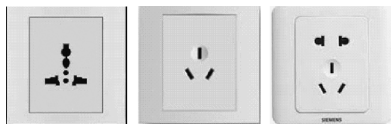

图 5-5　不同的插座接口

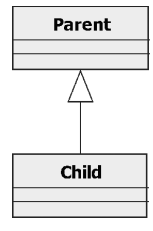

图 5-6　继承的 UML 图

语法

```
//Parent.java
package hust.inheritance;

public class Parent {
    //类的功能代码
}

//Child.java
```

```
package hust.inheritance;

public class Child extends Parent {
    //类的功能代码
}
```

Java 继承使用 extends 关键字连接父类和子类，Child 是子类，跟在子类名称之后的是父类。extends 后面只能使用一个类。

Parent 称为父类，Child 称为子类。

子类重用了父类的代码，具有了父类的接口服务，同时子类也可以扩充父类的功能。

Java 只支持单根继承。单根继承使程序代码简单，更容易理解，同时 Java 系统的垃圾回收器的实现变得容易许多。

5.4 父子关系

5.4.1 IS-A

如果子类继承父类而没有扩充功能的话，子类可以被完全当做一个父类看待，因为外在表现一致，这种关系称为 IS-A 关系，如图 5-7 所示。

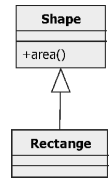

图 5-7 子类没有扩充父类功能

例 5-5 子类继承父类后没有扩充功能。

```
//Shape.java
package hust;

public class Shape {
    public float area() throws IllegalAccessException {
        throw new IllegalAccessException("没有实现此功能");
    }
}

//Rectangle.java
package hust;

public class Rectangle extends Shape {
    private float height;
    private float width;

    public Rectangle(float height,float width) {
        this.height = height;
        this.width = width;
    }

    public float area() throws IllegalAccessException {
        return height * width;
    }
}
```

Rectangle IS-A Shape。

一个矩形(Rectangle)由于继承了形状类(Shape)，便具有了父类提供的服务 area 功能，客户使用矩形，完全可以当做一个通用的几何形状来处理。

```
Shape rect = new Rectangle();                    //正确
```

这个语句可以这样描述：声明了一个矩形的几何形状。

> 即使父类和子类外在表现完全一致，父类也不能被当做子类来用。

```
Rectangle rect = new Shape();                    //错误
```

描述：声明了一个几何形状的矩形。这样的描述显然是错误的。如同说“人是张三”一样，它将大的概念变小了，这样是不正确的，反之“张三是一个人”这样的描述是正确的。

还有另外两种声明方法：

```
Shape shape = new Shape();                       //正确
```

描述：声明了一个几何形状的形状。

```
Rectangle rect = new Rectangle();                //正确
```

描述：声明了一个矩形形状。

实际上，现在很多编程语言已经支持在外在表现类似的情况下，可以被当做彼此的技术，有一个有趣的名字 Duck Typing；它是动态语言重要的特性之一。它描述了当一只鸟走路像鸭子，游泳像鸭子，叫起来也像鸭子，那么就认为它就是鸭子。

静态语言(Java/C#)类型的判定会在编译期进行，如果一系列类型需要对外界释放出某种共同的行为，那么它们则必须符合一个共同的协议(如基类或接口)。在支持 Duck Typing 的语言(如 JavaScript 或 Python)中，对于某个对象成员的访问会在运行时进行检查，所谓“延迟判定”。静态语言要想支持类似 Duck Typing 的技术，可以使用反射实现。Duck Typing 会带来很多好处，它将原有的代码变得更加通用，只要支持的行为相同，就可以执行。在语言层面上 Java 不支持 Duck Typing，只能通过反射实现类似的效果。

例 5-6 面向父类 Shape 编程。

```
//Client.java
public class Client {
    public void handle(Shape shape) {
        try {
            System.out.println(shape.area());
        } catch (IllegalAccessException e) {
            System.out.println(0);
        }
    }

    public static void main(String[] args) {
        Client client = new Client();
        Rectangle rect = new Rectangle(10,20);
        client.handle(rect);
    }
}
```

运行结果：

```
200.0
```

Client 中提供了一个服务 handle，用于处理形状，内部实现会调用 Shape 的 area 接口，handle 的参数可以接收所有被认为是 Shape 的类型，Rectangle 显然符合要求，这个 handle 函数支持所有基于 Shape 的子类型作为函数参数，非常通用。

handle 函数就是针对父类设计的，它具有一般性，如果将 handle 更改为例 5-7 所示：

例 5-7 针对具体类型编程。

```
public void handle(Rectangle rect) {
    try {
        System.out.println(rect.area());
    } catch (IllegalAccessException e) {
        System.out.println(0);
    }
}
```

handle 只能接收被认为是 Rectangle 的类型，不能支持椭圆、三角形等形状作为函数参数，这样的代码就不具有一般性，局限性很大，所以被更改的可能性也变大。

针对父类编程是一个好的习惯，它使代码更通用，更具一般性，带来了更大的扩展性，是推荐的设计方式。

形如：

```
Shape shape = new Rectangle();
```

它是一种针对父类的声明方式。

```
public void handle(Shape shape) {…}
```

这样的函数也是针对父类设计的函数，都是值得推荐的。

> 针对父类编程会带来另一个更大的好处：它引起了多态性。这是面向对象的核心技术之一。
>
> 针对父类编程的局限性就是：所有的对象都被当做了父类，即使子类有更多的扩充功能，也都无法使用，实现代码只能使用父类的通用性功能。

子类 Rectangle 提供了父类没有的特有功能：判断矩形是否是一个正方形(isSquare)，如图 5-8 所示。

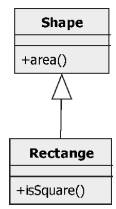

图 5-8 子类扩充父类的功能

```
Shape rect = new Rectangle();
rect.isSquare();                    //错误调用
```

这样声明的 rect 只具有父类的功能，它是一个几何形状，几何形状里并没有 isSquare 函数，所以不能直接调用。只能这样使用：

```
Rectangle rect = new Rectangle();
rect.isSquare();                    //正确
```

只有 rect 被当做 Rectangle 来声明的时候，它才具有 isSquare 行为。

从这个例子可以看出，针对父类编程的局限性，但是这个局限性却带来了一个好处就是：针对父类编写的代码，它是具有通用性的，它不是针对某一个具体类型编写的，这样实现的代

码所有的子类都适用。

```
public void handle(Shape shape) {
    try {
        System.out.println(shape.area());
    } catch (IllegalAccessException e) {
        System.out.println(0);
    }
}
```

此段代码无法调用 Rectangle 的特殊功能，但是此段代码却具有通用性。调用子类特有功能将使 handle 依赖特定子类。

不扩充子类的功能，那如果子类有特殊需求的功能怎么办？难道为了程序的通用性，就不去扩展子类的功能了吗？当然不是。子类扩展功能与否只是针对具体的情况而言。

> 尽可能不去扩充父类功能。这样的设计可以完全面向父类编程，使程序代码具有通用性。代码通用，意味着它具有一般化特点，适应大多数情况，扩展性强，代码被更改的可能性就小。

5.4.2 IS-LIKE-A

当子类扩充了父类的功能，子类提供的服务超过了父类提供的数量，二者外在表现并不完全一致，称这种关系为IS-LIKE-A，如图 5-9 所示。

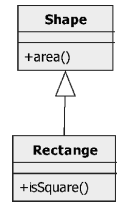

图 5-9 子类扩充父类的功能

例 5-8 子类继承父类后，扩充了自己的功能。

```
//Shape.java
package hust;

public class Shape {
    public float area() throws IllegalAccessException {
        throw new IllegalAccessException("没有实现此功能");
    }
}

class Rectangle extends Shape {
    private float height;
    private float width;

    public Rectangle(float height,float width) {
        this.height = height;
        this.width = width;
    }

    public float area() throws IllegalAccessException {
        return height * width;
    }

    public boolean isSquare() {
        return height == width;
    }
}
```

Rectangle IS-LIKE-A Shape。

> 在必要时，子类扩充父类功能，使子类实用性更强。但是带点来的缺点是：使代码通用性降低，僵化。二者取舍要根据实际情况慎重选择。

有很多可以在不更改父类接口功能的情况下扩展父类功能的方法，这些设计方法可以参考《设计模式》一书（Erich Gamma，Richard Helm，Ralph Johnson，John Vlissides. Design Patterns-Elements of Reusable Obiect-Oriented Software. 设计模式. 李英军，马晓星等译. 北京：机械工业出版社，2005.）。

5.5 向上转型

从继承的 UML 图 5-10 中可以看到，所有的 Shape 子类都可以被称为 Shape，这就是向上转型。从箭头的方向可以看到，向上转型是从子类向父类转换。

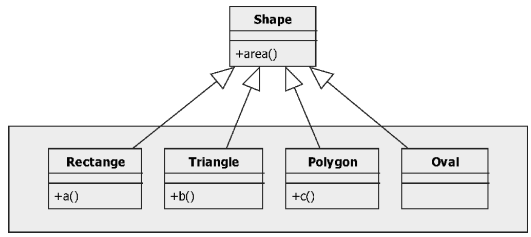

图 5-10 子类可以向上转型为父类

子类被当做父类来看，是非常安全的，因为子类至少拥有和父类一样的方法。客户使用这样向上转型的对象不会有任何问题。

```
Shape shape = new Rectangle();                    //正确
```

向上转型是安全的，是不需要强制转换的。

```
Shape shape = (Shape) (new Rectangle());
```

这样书写是没有必要的。

声明的类型（Shape）与实际创建的对象（Rectangle），二者就像 Shape 代理了 Rectangle 一样，客户只能通过 Shape 来间接操作后面的实际对象 Rectangle，Shape 所表现的服务接口就是 Rectangle 的接口，客户看不到除 Shape 接口以外其他的行为。Rectangle 行为再丰富，也只能通过 Shape 向外表现，特有的行为全部被 Shape 阻挡在后端，客户根本看不到。但是实际上后台的 Rectangle 实实在在地存在，全部特有的、不能调用的方法全部存在。

> 向上转型带来的直接问题就是：转型后，子类特有的方法全部丢失。但是这些方法可以通过向下转型找回来，因为它们虽然从父类的角度是看不见了，但实际上它们都存在于内存当中，并没有消失。
>
> 向下转型是不安全的，客户看到的父类 Shape 所对应的后台真实对象有很多（Rectangle、Triangle 等），使用者不能够判断一个 Shape 后端到底是什么。

```
public void show(Shape shape) {
    Rectangle rect = (Rectangle)shape;      //可能引发错误
}
```

函数 show 的参数在运行时，函数并不知道传递过来的 shape 到底是什么，不能主观地认为是一个 Rectangle，就进行强制向下转型为 Rectangle，万一是一个 Triangle 类型，程序将会出错。所以向下转型是不安全的，但是如果能够判断传递过来的 shape 类型是哪个子类型的话，强制向下转型是不会有问题的。

```
public void show(Shape shape) {
    if(shape instanceof Rectangle) {        //先判断是否是需要转换的类型
        Rectangle rect = (Rectangle)shape;
        rect.a();
    }
}
```

在强制将 shape 转换为子类时，先判断所指向的对象是否是需要转换的类型，然后再转换，可以防止错误的向下转型。

instanceof 可以判断给定的一个对象是否是指定的类型，判断结果是 true 或者 false。

向下转型后，所有子类的特有方法都表现出来了，但是这样一段通用的代码依赖子类，导致所有使用此段代码的客户都需要引入 Rectangle 类型，即使不需要，也需要带着，这是客户很不愿意看到的。就像到商店买一根铅笔，但是售货员说：由于铅笔使用了木头，你必须把一棵树也买走一样，那将是多么无奈。

总结如下：

(1) 针对父类编写的代码，自然而然地使用了向上转型。所有的子类被传递给函数参数时，自然转换为父类。

(2) 针对父类编写的代码，向下转型是使用子类特有方法的唯一手段。

(3) 利用向上转型编写的代码具有一般性，通用性好。

(4) 利用向下转型使用子类特有方法的代码丰富了函数功能，但具有特殊性，不具有一般性，导致了代码对子类的依赖。

所以从向上转型的优点来看，在通用代码或类库的设计上，尽可能地减少向下转型是必要的。对于客户编写的、有目的的特殊代码，向下转型提供了丰富的子类行为，为完成复杂任务提供了方便。

5.6 子类的初始化

例 5-9 实例化子类的初始化顺序。

```
//Shape.java
package hust;

public class Shape {
    public Shape() {
        System.out.println("Shape");
    }
}
```

```
//Rectangle.java
package hust;

public class Rectangle extends Shape {
    public Rectangle() {
        System.out.println("Rectangle");
    }

    public static void main(String[] args) {
        Rectangle rect = new Rectangle();
    }
}
```

运行结果：

```
Shape
Rectangle
```

从结果来看，实例化子类，首先调用了父类的构造函数，初始化了父类，然后调用了子类的构造函数，初始化了子类。说明在内存空间里，实例化一个子类，给子类分配空间之前，先给父类分配了空间。

可以简单形象的理解为：

(1) 实例化子类后，在内存空间中的对象分布如图 5-11 所示。

(2) 如果子类完全没有扩展父类的功能，那么它们的大小一致，如图 5-12 所示。

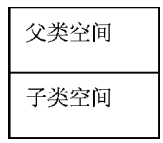

图 5-11　父类和子类实例化的空间分布示意图

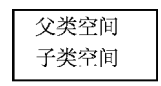

图 5-12　子类不扩充父类时的空间分布示意图

5.6.1　无默认构造函数

例 5-10　父类无默认构造函数，子类只有默认构造函数，无法实例化子类。

```
//Shape.java
package hust;

public class Shape {
    public Shape(int color) {
        System.out.println("Shape");
    }
}

//Rectangle.java
package hust;

public class Rectangle extends Shape {
```

```
    public Rectangle() {                    //错误
        System.out.println("Rectangle");
    }

    public static void main(String[] args) {
        Rectangle rect = new Rectangle();
    }
}
```

如果父类没有提供默认的构造函数,而只提供了一个非默认构造函数的话,上述代码是不正确的。子类在继承了父类之后,自己编写了一个默认构造函数,实际上也就是要求子类在实例化时,调用默认构造函数。可是,由于创建子类,必先实例化父类,而父类没有默认构造函数可调用,所以造成了错误。

```
Rectangle rect = new Rectangle();
```

只有这一句指示调用默认构造函数实例化,无法再次指定父类调用自己的特定构造函数。

> 实例化子类时,如果没有明确指出调用哪个构造函数实例化父类,那么,系统会自动调用父类的默认构造函数进行初始化。

这样的代码也是不正确的:

```
//Shape.java
package hust;

public class Shape {
    public Shape(int color) {
        System.out.println("Shape");
    }
}

//Rectangle.java
package hust;

class Rectangle extends Shape {
    public static void main(String[] args) {
        Rectangle rect = new Rectangle();
    }
}
```

子类提供了默认构造函数,而父类没有,所以无法正确实例化父类。

例 5-11 这样的代码是正确的。

例 5-11 实例化子类时,默认调用父类的默认构造函数。

```
package hust;

public class Shape {
    public Shape() {
        System.out.println("Shape");
    }
}
```

```
class Rectangle extends Shape {
    public Rectangle(int height,int width) {
        System.out.println("Rectangle");
    }

    public static void main(String[] args) {
        Rectangle rect = new Rectangle(10,20);
    }
}
```

在子类没有明确提出要调用父类默认构造函数初始化父类时,系统会自动调用父类的默认构造函数进行初始化父类,而父类提供了这样的默认构造函数,所以可以顺利地被初始化。

如果父类不存在默认构造函数:

```
package hust;

public class Shape {
    public Shape(int color) {
        System.out.println("Shape");
    }
}
```

那么系统无法知道该调用哪个构造函数初始化父类。如果子类明确指出调用构造函数初始化父类的话,这样的代码就是正确的。

例 5-12 子类构造函数显示调用父类指定构造函数实例化父类。

```
//Shape.java
package hust;

public class Shape {
    public Shape(int height) {
        System.out.println("Shape");
    }
}

//Rectangle.java
package hust;

class Rectangle extends Shape {
    public Rectangle(int height,int width) {
        super(10);                          //显式调用父类的构造函数初始化父类
    }
    public static void main(String[] args) {
        Rectangle rect = new Rectangle(10,20);
    }
}
```

代码使用了 super 关键字显式调用了父类的构造函数初始化父类,这样父类即使没有默认构造函数也没有任何问题,也能够被顺利地初始化,super 关键字只能放到子类构造函数的第一行调用,而且只能在构造函数中使用,确保在子类构造之前先构造父类。super 关键字后面会详细解释。

这一小节介绍了在初始化子类之前必须先初始化父类的注意事项,子类要想顺利地完成

初始化，必须保证能够顺利地构造父类，如果子类没有明确指出父类的初始化构造函数，默认是调用父类默认构造函数。

5.6.2 初始化顺序

在类的初始化一节，讲述了类中成员变量的初始化顺序问题，如果加入子类，这个顺序将会更加有趣。

例 5-13 一个全面测试子类初始化顺序的示例。

```
//Shape.java
package hust;

public class Shape {
    private Tag tag1 = new Tag("p1");
    private static Tag tag2 = new Tag("p2");
    private Tag tag3 = new Tag("p3");
    private static Tag tag4 = new Tag("p4");

    public Shape() {                        //构造函数
        Tag tag5 = new Tag("p5");
    }

    public Tag tag6 = new Tag("p6");
    public static Tag tag7 = new Tag("p7");
}

class Rectangle extends Shape {
    private Tag tag1 = new Tag("c1");
    private static Tag tag2 = new Tag("c2");
    private Tag tag3 = new Tag("c3");
    private static Tag tag4 = new Tag("c4");

    public Rectangle() {                    //构造函数
        Tag tag5 = new Tag("c5");
    }

    public Tag tag6 = new Tag("c6");
    public static Tag tag7 = new Tag("c7");

    public static void main(String[] args) {
        Shape shape = new Rectangle();
        System.out.println();
        Shape shape1 = new Rectangle();
    }
}
class Tag {
    public Tag(String str) {
        System.out.print(str + ",");
    }
}
```

为了直观，父类 p 前缀，子类 c 前缀。

运行结果：

```
p2,p4,p7,c2,c4,c7,p1,p3,p6,p5,c1,c3,c6,c5,
p1,p3,p6,p5,c1,c3,c6,c5,
```

从这个结果可以看出先初始化了：

(1) 父类的静态变量(按声明顺序)：p2、p4、p7。

(2) 子类的静态变量(按声明顺序)：c2、c4、c7。

(3) 父类的成员变量(按声明顺序)：p1、p3、p6。

(4) 父类的构造函数：p5。

(5) 子类的成员变量(按声明顺序)：c1、c3、c6。

(6) 子类的构造函数：c5。

(7) 再次实例化子类，静态变量不再进行初始化工作，只初始化非静态变量，从第(3)步重复执行。

从这个顺序可以看出：所有的工作都是为了保证子类能够被正确初始化，子类要有一个坚实的基础(父类)才能够被成功创建。

(1) 子类依赖父类。

(2) 子类对象依赖成员变量(包括父类的)。

> 静态变量的特性永远都是最先调用，并只调用一次，它不依赖对象的存在而存在。

其余的所有成员变量都可能在子类实例化时被调用，所以必须保证首先初始化完成，而且顺序是保证父类创建完成之后，才创建子类。子类在有了父类这个良好基础的前提下，才能被顺利地实例化。

5.6.3　protected 关键字

访问权限：本类、派生的子类、相同包中的其他类。

例 5-14　通过不同包的客户类来测试 protected 权限的可访问性。

```
//Parent.java
package hust.inheritance;

public class Parent {
    protected int age = 10;
    public int getAge() {
        return age;
    }
}

//Child.java
package hust.inheritance;                    //相同包

public class Child extends Parent {          //子类
    public void setAge() {
        age = 20;                            //子类可以直接访问父类的 protected 变量
    }
}
```

```
//OtherClient.java
package hust.inheritance;                          //相同包

public class OtherClient {
    public void visitAge() {
        Parent parent = new Parent();
        parent.age = 100;                          //正确,相同包中的其他类可以访问 protected 变量
    }
}

//Client.java
package hust.other;                                //不同包
import hust.inheritance.Parent;

public class Client {
    public void visitAge() {
        Parent parent = new Parent();
        parent.age = 100;                          //错误,不同包中的类不可以访问 protected 变量
        System.out.println(parent.getAge());//只能调用公有方法
    }
}


//OtherChild.java
package hust.other;                                //不同包中的子类
import hust.inheritance.Parent;

public class OtherChild extends Parent {
    public void setAge() {
        age = 20;                                  //不同包的子类也可以访问父类 protected 变量
    }
}
```

使用 protected 最恰当的方式就是在父子关系中,它能够保证一个家族(父子关系)中成员能够顺利访问,而阻止了外界的非法访问。其次才是一个包中的其他类的访问,通常为了有效组织类,使用了包技术,它保证了一系列相关的类能够放到一个包中,它们彼此或多或少有些关系,这样使用 protected 修饰的元素,可以被同包中其他类型访问能够在实现具体逻辑功能上带来方便,protected 给同包中其他类带来了一些访问特权,外包中的类没有这个访问特权。

> protected 只能在自己的继承序列上访问,两个并列继承序列上的类型不可互相访问。

如图 5-13 所示,Child 可以访问 Parent 中的 protected 元素,由于 Child 和 OtherChild 没有继承关系,所以 Child 是不能访问 OtherChild 中的 protected 元素的。

注意:如图 5-14 所示,ChildChild 在 b 包中,它继承于 a 包中的 Child,而 Child 继承于 Parent,与 Parent 不在一个包中。

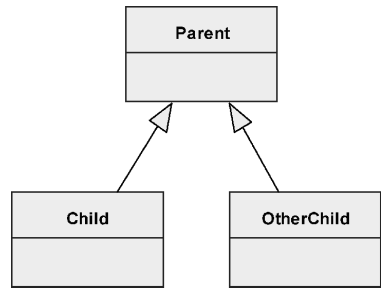

图 5-13 并列的两个子类 Child 和 OtherChild

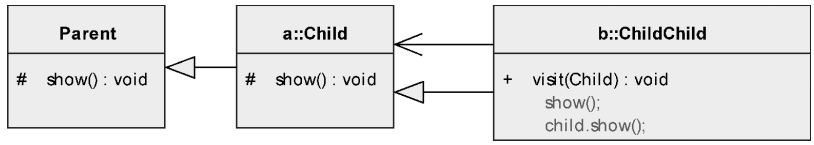

图 5-14 分布在不同包中的继承关系

例 5-15 分布在不同包中的继承关系类型 protected 的可访问性。

```
//Parent.java
package hust.pro;

public class Parent {
    protected void show() {
        System.out.println("parent");
    }
}

//Child.java
package hust.pro.a;
import hust.pro.Parent;

public class Child extends Parent {
    protected void show() {
        System.out.println("child");
    }
}

//ChildChild.java
package hust.pro.b;
import hust.pro.a.Child;

public class ChildChild extends Child {
    protected void visit(Child child) {
        show();                          //正确
        child.show();                    //错误
    }
}
```

注意 ChildChild 和 Child 不在同一个包中，ChildChild 继承了 Child，它自然可以在 visit 函数中访问继承过来的 show 函数，但是注意，这个 visit 函数中参数是一个 Child 类型，函数中调用了参数 Child 对象的 show 函数，这样做，编译会产生错误！protected 修饰的元素只能在父子关系中使用，而此时，visit 参数中的 Child 对象和当前的 ChildChild 并不是父子关系，而是关联关系，关联关系将会受到 protected 关键字访问权限的限制：非同包中的类 ChildChild 是不能访问 Child 的 protected 元素的。

试想一下如果 child.show 是正确的，会出现什么结果？那就是 ChildChild 可以访问与其并列的其他子类的受保护的成员。这可能对于被使用者是一个灾难。因为它们失去了 protected 的保护，显然那将是一个严重的漏洞。

5.6.4 final 的使用

> final 关键字修饰的元素表示不可修改之意。

例 5-16 使用 final 定义一个不可修改的变量。

```
//FinalClazz.java
package hust.finalsample;

public class FinalClazz {
    public final int age = 10;

    public static void main(String[] args) {
        FinalClazz finalClazz = new FinalClazz();
        finalClazz.age = 20;              //错误,final 变量不可修改
    }
}
```

> final 可以修饰很多元素,都表示不可更改之意。
> (1) 修饰变量。表示变量值不可修改。
> (2) 修饰类。表示该类不可继承。
> (3) 修饰函数。表示函数不可以被重写。
> (4) 修饰函数参数。表示参数值不可以修改。

static 具有唯一一份之意,final 具有不可更改之意,二者联合使用可以定义一个常量。

例 5-17 使用 static 和 final 联合定义一个常量。

```
package hust.finalsample;

public class FinalClazz {
    public static final int AGE = 10;

    public static void main(String[] args) {
        System.out.println(FinalClazz.AGE);//常量
    }
}
```

1. final 修饰类

```
package hust.finalsample;

public final class FinalClazz {
}

class FinalChildClazz extends FinalClazz{  //错误,final 类不可以被继承

}
```

字符串类 String 就是一个 final 类型,它是不可以被继承的!

2. final 修饰函数

```
package hust.finalsample;

public class FinalClazz {
    public final void finalMethod() {
    }
```

```
}

class FinalChildClazz extends FinalClazz{
    public void finalMethod() {            //错误,final 函数不可以被子类重写
    }
}
```

私有函数本身就是不可以被重写的,所以私有函数声明为 final 是多余的。

3. final 修饰函数参数

```
package hust.finalsample;

public class FinalClazz {
    public void show(final int age) {
        age = 20;                          //错误,final 参数不可以被更改
        System.out.println(age);
    }
}
```

从这些例子都可以看出 final 有不可更改之意。所以,修饰的元素必须有确定的值,也就是必须声明的同时要初始化完成,因为之后没有机会再次赋值,一旦可以再次赋值,就表示可以修改之意,这与 final 强调的不可修改相悖。

```
package hust.finalsample;

public class FinalClazz {
    private final int age;                 //错误,声明的时候没有被初始化值
}
```

age 变量声明后没有赋初始值,虽然成员变量系统会自动初始化为 0,但是此时 final 修饰,必须显式初始化。

```
package hust.finalsample;

public class FinalClazz {
    private final int age = 10;            //正确
}
```

变量是类时:

```
package hust.finalsample;

public class FinalClazz {
    private final Tag tag = new Tag();

    public void show() {
        tag = new Tag();                   //错误,不可以修改 final 类型的 tag 值
    }
}

class Tag{
    public int age = 10;
}
```

然而,这样是正确的:

```
package hust.finalsample;

public class FinalClazz {
    public void show(final Tag tag) {
        tag.age = 30;                          //正确,可以修改 tag 对象中的值
        System.out.println(tag.age);
    }
}

class Tag{
    public int age = 10;
}
```

由于 final 强调的是变量值不可更改,如果声明的变量是一个类类型(非主类型):

```
public final Tag tag = new Tag();
```

tag 变量是一个类类型,tag 的值是指向 Tag 对象的首地址,也就是说这个地址指向不能改变。更改 Tag 对象中的值,实际上并没有改变对象的首地址,所以符合 final 修饰的要求。但是

```
tag = new Tag();                               //错误
```

重新给 tag 指定了一个新的对象,tag 的值(对象首地址)改变了,这样的代码是错误的。

数组在 Java 中是非主类型,所以以下代码是正确的:

```
package hust.finalsample;

public class FinalClazz {
    public FinalClazz(final int[] tags) {
        tags[0] = 100;
    }
}
```

改变数组元素的值,不会改变数组对象的首地址。

可以看出:

> final 在修饰类时,并不限制用户修改对象包含的变量值,只是限制了对象的转移,只能针对某一个对象进行操作,中途不可更改对象。

4. 空白 final

```
package hust.finalsample;

public class FinalClazz {
    private final int age;                     //声明后,没有赋初始值
    public FinalClazz() {
        age = 10;                              //final 变量,在构造函数里延后初始化
    }
}
```

这种延后初始化 final 变量的特例,实际上和声明马上初始化相似,它带来了一定的灵活

性，同时又保证了变量在使用之前值是确定的。

由于 final 修饰类时阻止了继承，所以在设计 final 类时要慎重，这样的决定可以导致使用者无法通过继承复用代码，但是通过组合方式是可以重用代码的。

Java 系统类库中有很多 final 类型，例如 String 类就被定义为 final 类，所以，String 是不可以被继承、派生子类型的，只能通过组合复用其中的代码。

5.7 重写父类的方法

5.7.1 如何重写

重写覆写(Override)和重载(Overload)有些类似，都是针对同名函数的一些操作，但它们有着本质的区别。

由于子类通过继承自动拥有了父类的方法，但是由于具体需求不同，继承来的父类方法并不一定适合子类。那该如何处理呢？

(1) 在子类中重新定义一个新的方法，这是在扩充子类功能。

(2) 在子类中重新定义这个父类方法的功能，这就是子类重写父类方法。

由于重写了父类方法，在运行时，子类会表现出和父类不同的行为特点。也就是说，即使调用相同的方法，由于具体运行时所对应的对象不同，所表现出来的行为也不同，这就是面向对象中最为重要的核心技术：多态性。

> 重写发生在子类中，它是针对父类相同签名函数的重定义。
> 重载是发在一个类中，它是一个类中的相同函数名称的不同函数的重定义。

重载概念只是针对同名的不同函数。

例 5-18 定义一个重载函数。

```
//Parent.java
package hust.inheritance;

public class Parent {
    public void show() { }
    public void show(int age) {}//重载
}
```

show 函数名相同，但是签名并不相同，称之为重载。

例 5-19 在子类中定义一个重载函数。

```
//Parent.java
package hust.inheritance;

public class Parent {
    public void show() { }
}
//Child.java
package hust.inheritance;

public class Child extends Parent {
    public void show(int age) {}//重载
}
```

存在于父类和子类中，同名的不同函数依然称之为重载。

重写概念存在于父类与子类之间针对相同签名的函数。

例 5-20 子类自动继承父类拥有的函数。

```
//Parent.java
package hust.inheritance;

public class Parent {
    public void show() {
        System.out.println("parent");
    }
}

//Child.java
package hust.inheritance;

public class Child extends Parent {
    public static void main(String[] args) {
        Child child = new Child();
        child.show();
    }
}
```

子类在继承了父类之后，自然拥有了父类 show 的功能。

运行结果：

```
parent
```

例 5-21 子类重写父类函数。

```
//Parent.java
package hust.inheritance;

public class Parent {
    public void show() {
        System.out.println("parent");
    }
}

//Child.java
package hust.inheritance;

public class Child extends Parent {
    public void show() {                    //重写父类 show 函数
        System.out.println("child");
    }
    public static void main(String[] args) {
        Child child = new Child();
        child.show();
        Parent child2 = new Child();
        child2.show();
    }
}
```

由于某些原因，父类的 show 功能并不符合子类的要求，所以子类重写了 show 功能。此时再次调用子类函数 show 时：

运行结果：

```
child
child
```

可以看到，由于子类重新定义了 show 函数后，打印的结果是期望的值“child”。

这种子类重新定义父类函数的技术称为重写。它可以让相同的函数在调用时呈现出不同的效果，这将在 5.9 节中重点介绍。

> 重写的关键在于子类重写的函数要和父类签名一致，否则就不是重写，那是子类扩充了父类的功能。

例 5-22 重写父类函数时函数的返回值不同，返回值没有任何继承关系。

```
class Parent {
    public A f() {
        return new A();
    }
}
class Child extends Parent {
    public C f() {
        return new C();
    }
}

class A {}
class B extends A {}
class C {}
```

这段代码，子类 Child 定义的 f 函数虽然和父类的签名一致，但是返回值不同，一个是 A，一个是 C，C 和 A 没有关系，是独立的两个类型，这会导致子类中出现了同一个函数调用，返回了不同的类型值，这是错误的，这不是重写。

在编译器就会出现错误提示：

> hust.inheritance.Child 中的 f()无法覆盖 hust.inheritance.Parent 中的 f()；正在尝试使用不兼容的返回类型
>
> 找到：hust.C
>
> 需要：hust.A

但是，子类再重写父类函数时，可以返回比父类更具体的类型，这样做是安全的。

例 5-23 重写父类函数时函数返回值不同，返回值之间有继承关系。

```
class Parent {
    public A f() {
        return new A();
    }
```

```
}
class Child extends Parent {
    public B f() {
        return new B();
    }
}
class A {}
class B extends A {}
```

注意子类重写父类 f 函数时，返回值和父类并不一致，一个是 A，一个是 B，和前面那个错误的例子不同，A 和 B 是有关系的，B 是 A 的子类，B 实际上要比 A 更具体，B 可以被完全当做 A 来看待，所以即使同一个函数在调用时都当做 A 类型看来也不会有任何问题。如果被当做 B 看来也不会有问题，因为那个时候内存中肯定是 Child 对象。

5.7.2 equals 函数

> Java 中，一个类如果没有明确指定父类，默认是隐式继承于 Object 类。

```
public class Parent {
}
```

Parent 类没有明确指出继承哪个类，实际上此时它是隐式地继承于 Object 类，等同于

```
public class Parent extends Object {
}
```

Object 默认作为所有类的基类，它定义了所有类共有的特性和功能，如图 5-15 所示。
可以重写的有以下函数，如图 5-16 所示。

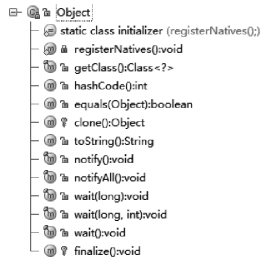

图 5-15 Object 类

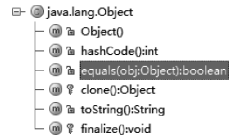

图 5-16 Object 类中可重写的函数

例如常用的 equals 函数，它是比较两个对象是否相等。

两个数字相互比较，直接使用==符号即可，但是如果两个对象相比较，使用==符号比较的概念将不明确。例如“两个人，他们相等”这样一句话非常不明确，到底两个人什么相等，这段话没有明确指出。所以比较两个对象要明确具体比较什么。例如，两个人，他们的身高相同、体重相同、肤色相同等。比较的概念要明确，否则无法比较。在 Java 中对象的比较也类似，必须明确指定所要比较的具体属性是什么。

Java 中对象使用“==”是比较变量的值(对象的引用地址)，而并不是变量所引用的对象本身，equals 才是比较两个对象是否相同的正确方法。当比较值是主类型时，变量本身的值就是实际比较的值，那么“==”比较是正确的。

如图 5-17 所示，a 和 b 变量中的值是所指向对象的首地址，所以使用＝＝比较的结果显然不会相等，只有使用 equals 函数才能做到所引用对象的比较，但是具体的比较方法需要重写对象的 equals 函数。

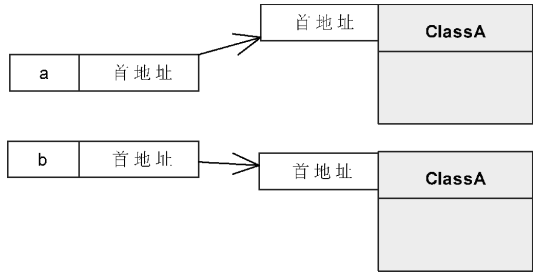

图 5-17　不同的变量指向同一个类的不同对象

例 5-24　使用＝＝和 equals 比较对象。

```
package hust.equalsample;

public class EqualClazz {
    private int age;
    public void setAge(int age) {
        this.age = age;
    }

    public static void main(String[] args) {
        EqualClazz equalClazz1 = new EqualClazz();
        EqualClazz equalClazz2 = new EqualClazz();
        System.out.println(equalClazz1 == equalClazz2);          //false
        System.out.println(equalClazz1.equals(equalClazz2));     //false
    }
}
```

该段代码使用了＝＝和 equals 函数进行对象比较。

运行结果：

```
false
false
```

使用＝＝进行对象比较，由于两个对象在内存中的地址不相同，变量 equalClazz1 和 equalClazz2 中的值自然不相同。第二个 equals 函数比较默认使用了 Object 中的 equals 函数，实际上等同于使用了＝＝。

Object 中 equals 的代码是：

```
Object 中 equals 的代码是:
public boolean equals(Object obj) {
    return (this == obj);
}
```

要想正确表达两个对象的比较，应该重写父类的 equals 函数，明确指出相等的条件。

例 5-25 重写 equals 函数。

```
//EqualClazz.java
package hust.equalsample;

public class EqualClazz {
    private int age;

    public void setAge(int age) {
        this.age = age;
    }

    public boolean equals(Object obj) {
        if(obj instanceof EqualClazz)
        {
            EqualClazz equalClazz = (EqualClazz) obj;
            return equalClazz.age == age;
        }
        return false;
    }

    public static void main(String[] args) {
        EqualClazz equalClazz1 = new EqualClazz();
        EqualClazz equalClazz2 = new EqualClazz();
        System.out.println(equalClazz1 == equalClazz2);         //false
        equalClazz1.setAge(10);
        equalClazz2.setAge(10);
        System.out.println(equalClazz1.equals(equalClazz2));    //true
        equalClazz2.setAge(20);
        System.out.println(equalClazz1.equals(equalClazz2));    //false
    }
}
```

此段代码重写了父类 equals 函数，在比较之前，先判断函数的参数是否是 EqualClazz 类型，如果不是，直接返回 false，告知调用者二者不相等，如果是和本类相同的对象，那么进行强制转换为 EqualClazz，然后判断其中的 age 变量是否相同。

运行结果：

```
false
true
false
```

可见，在比较 EqualClazz 对象的核心是在比较 age。比较的信息非常明确。

> 判断两个对象是否相同，应该使用 equals 函数，而不是==。

5.7.3 toString 函数

toString 函数也是一个重要的、需要重写的 Object 函数。它是打印对象时默认调用的函数。Object 的 toString 函数定义：

```
public String toString() {
    return getClass().getName() + "@" + Integer.toHexString(hashCode());
}
```

它返回了对象的完全限定名和 hash 码字符串。例如例 5-26。

例 5-26 打印一个对象。

```
//Person.java
package hust.equalsample;

public class Person {
    private int age = 0;
    public void setAge(int age) {
        this.age = age;
    }

    public static void main(String[] args) {
        Person person = new Person();
        System.out.println(person);
    }
}
```

运行结果：

```
hust.equalsample.Person@ca0b6.
```

相当于 System.out.println(person.toString());这样的打印结果并不友好。

例 5-27 通过重写 toString 可以自定义打印对象。

```
package hust.equalsample;

public class Person {
    private int age = 0;
    public void setAge(int age) {
        this.age = age;
    }

    public String toString() {
        return "Person Age = " + age;
    }

    public static void main(String[] args) {
        Person person = new Person();
        System.out.println(person);
    }
}
```

运行结果：

```
Person Age = 0
```

打印结果很友好。同 equals 类似，如果单纯说把这个对象打印出来，这样的话语非常宽

泛，也就无法打印出友好的结果。必须明确指出要打印什么，打印的结果才具有实际意义。

通过上述几个实用的例子可以看到重写在编程中的重要作用，子类经常会根据具体情况重写父类提供的方法。重写之后，它们会根据具体的情况被正确调用。

5.8 super 引用

super 关键字和 this 关键字有些类似。

super 关键字在子类中使用，代表父类对象，通过 super 可以调用父类的变量和方法。

5.8.1 super 调用父类构造函数

上面章节讲述了子类在初始化时，在没有明确指出调用哪个构造函数初始化父类时，系统会自动调用父类的默认构造函数，同时也可以显式地调用父类构造函数进行父类的初始化工作。

例 5-28 显式调用父类构造函数实例化父类。

```
package hust;

public class Shape {
    public Shape(int height) {
        System.out.println("Shape");
    }
}

class Rectangle extends Shape{
    public Rectangle(int height,int width) {
        super(10);                              //显式调用父类的构造函数初始化父类
    }
    public static void main(String[] args) {
        Rectangle rect = new Rectangle(10,20);
    }
}
```

super 调用父类构造函数时，直接代表父类对象，使用方法：

(1) 调用父类默认构造函数：super()；

(2) 调用非默认构造函数，形如 super(10)，super("hello")。

这个与调用本类自己的构造函数 this 有些类似。

5.8.2 this 调用本类构造函数

1. this 调用本类默认构造函数

```
package hust.inheritance;

public class Parent {
    public Parent() {

    }
    public Parent(int age) {
        this();                                 //初始化时调用本类默认构造函数
```

```
    }
}
```

2. this 调用非默认构造函数

```
package hust.inheritance;

public class Parent {
    public Parent() {
        this(10);                          //调用本类非默认构造函数
    }
    public Parent(int age) {
    }
    public void show() {
        System.out.println("parent");
    }
}
```

> super 和 this 在调用构造函数时,调用语句必须在构造函数的第一行。

```
public Parent() {
    age = 10;
    this(10);                              //错误,因为调用语句不在第一行
}
//
public Parent(int age) {
    this.age = age;
    this();                                //错误,因为调用语句不在第一行
}
```

5.8.3 super 调用父类变量

如果在子类中不存在与父类冲突的变量或者函数,没有必要显式使用 super 调用。

例 5-29 不存在变量覆盖问题时,不需要使用 super 或者 this。

```
//Parent.java
package hust.inheritance;

public class Parent {
    protected int age = 10;
    public void show() {
        System.out.println("parent");
    }
}

//Child.java
package hust.inheritance;

public class Child extends Parent {
    public void show() {
        System.out.println(super.age);     //打印 10
        System.out.println(this.age);      //打印 10
        System.out.println(age);           //打印 10,此处三个语句同等含义
```

```
    }

    public static void main(String[] args) {
        Child child = new Child();
        child.show();
    }
}
```

运行结果：

```
10
10
10
```

age 变量被设置为 protected，子类可以访问到，在子类 show 函数中调用，没有任何歧义存在，整个类的继承关系中只有一个 age 变量。所以，子类 show 函数中的三个语句是一个意思，建议直接使用第三个调用语句。

但是如果在子类中也存在一个 age 变量，它覆盖了父类的 age 变量，此时就需要显式调用了。如果子类被定义为如例 5-30 所示。

例 5-30 子类变量覆盖了父类变量时，需要显式使用 super 或者 this。

```
//Child.java
package hust.inheritance;

public class Child extends Parent {
    protected int age = 20;
    public void show() {
        System.out.println(super.age);      //打印 10
        System.out.println(this.age);       //打印 20
        System.out.println(age);            //打印 20
    }

    public static void main(String[] args) {
        Child child = new Child();
        child.show();
    }
}
```

运行结果：

```
10
20
30
```

由于子类也定义了一个 age 变量，默认系统使用的是本类的 age 变量，调用父类的 age 变量就需要使用 super 关键字显式调用了。

5.8.4 super 调用父类函数

函数的调用与变量类似，只有在子类中重写了父类的函数时，需要显式使用 super 关键字。

例 5-31 使用 super 显式调用被重写的父类函数。

```
//Parent.java
package hust.inheritance;

public class Parent {
    public void show() {
        System.out.println("parent");
    }
}

//Child.java
package hust.inheritance;

public class Child extends Parent {
    public void show() {
        System.out.println("child");
        super.show();
    }

    public static void main(String[] args) {
        Child child = new Child();
        child.show();
    }
}
```

运行结果：

```
child
parent
```

通过 super 调用父类变量和函数，可以在任意位置调用，没有像调用父类构造函数那样，强制要求必须在第一行调用。

从上述几个例子中也可以看出：不管是否存在同名覆盖或者重写问题，父类元素（变量与函数）与子类元素在内存中同时存在。

5.9 多态性

5.9.1 多态的定义

> 多态(Polymorphism)：在运行时刻接口匹配的对象能互相替换的能力，指的是运行时多态。

这里重点是：

(1) 替换行为发生在运行时刻。

(2) 运行时刻要有能力判断对象接口是否匹配。

(3) 替换后，调用替换对象的方法。

这几点是面向对象语言支持多态的基础，要实现这个能力，需要“后期绑定机制”。

后期绑定机制使系统能够在运行时判断对象类型，从而调用恰当的方法执行。编译期程序并不知道运行时刻传递过来的对象是什么，所以必须要有能力动态判断对象类型，然后进行

替换。

例 5-32 运行期才知道传递的实际运行对象是什么。

```
//LateBinding.java
package hust.latebinding;

public class LateBinding {
    public void show(Shape shape) {
        System.out.println(shape.area());
    }

    public static void main(String[] args) {
        LateBinding lateBinding = new LateBinding ();
        Shape shape = new Rectangle();
        lateBinding.show(shape);
    }
}
```

这样一个 show 函数，它在编译期不会知道运行时传递给它的是一个 Rectangle 类型，只有在运行的时候才能确切知道。main 函数里虽然已经明确写出了传递的是一个 Rectangle，但是 show 函数并不知道。

例 5-33 通过输入动态创建的运行期对象。

```
//LateBinding.java
package hust. latebinding;

public class LateBinding {
    public void show(Shape shape) {
        System.out.println(shape.area());
    }

    public static void main(String[] args)
    throws ClassNotFoundException,IllegalAccessException,InstantiationException {
        LateBinding lateBinding = new LateBinding ();
        Shape shape = (Shape) Class.forName(args[0]).newInstance();
        lateBinding.show(shape);
    }
}
```

所有创建的 Shape 类型都是通过用户输入进行动态创建的，这样 show 函数在编译期无论从哪里都是不可能知道运行期传递给它的是什么类型的 Shape，都需要后期动态绑定机制动态判断对象类型，进行向上转型。

```
public void show(Shape shape) {
    System.out.println(shape.area());
}
```

从这个函数定义可以看到，程序调用了 Shape 的 area 函数完成用户需求，也仅仅表达了这个含义。但是在运行时，所有 Shape 类型（Rectangle，Triangle 等）都符合这个函数的参数类型约定，参数向上转型确实使子类类型信息丢失，show 函数将所有的对象都当做 Shape 对象看待。

如果系统永远都是只调用 Shape 类的 area 函数，那将很令人遗憾。用户期望的是调用传递的 Rectangle 类型的方法，程序就应该自动调用 area 重写的函数。这是一个很自然的行为，就应该这样调用，而不能永远只调用所声明的 Shape 类型的 area 函数。

令人兴奋的是，Java 的后期绑定机制完成了用户的期望，运行时确实是调用了传递给函数的可替换对象的行为。

例 5-34 重写父类函数后表现出的多态性。

```
//Shape.java
package hust.bind;

public class Shape {
    public int area() {
        return 0;
    }
}

//Rectangle.java
package hust.bind;

public class Rectangle extends Shape {
    private int height;
    private int width;
    public Rectangle(int height,int width) {
        this.height = height;
        this.width = width;
    }

    public int area() {
        return height * width;
    }
}

//Client.java
package hust.bind;

public class Client {
    public void show(Shape shape) {
        System.out.println(shape.area());
    }

    public static void main(String[] args) {
        Client client = new Client();
        Shape rect = new Rectangle(10,20);
        client.show(rect);
    }
}
```

运行结果：

```
200
```

显然，show 函数调用了 Rectangle 的重写之后的 area 函数，忽略了 Shape 的 area 函数。用户也期望这样的结果。

5.9.2 实现多态的基本步骤

通过上面的知识，构成多态已经变得非常清晰：

> (1) 需要继承。
> (2) 子类重写父类方法。
> (3) 接口设计针对父类编程。
> (4) 客户创建具体的子类传递给接口。

例 5-35 多态的实现。

```
//Parent.java
public class Parent {
    public void show() {
        System.out.println("parent");
    }
}
//Child.java
public class Child extends Parent {            //继承
    public void show() {
        System.out.println("child");           //重写父类方法
    }
}
//Client.java
public class Client {
    public void call(Parent arg) {             //针对父类编程
        arg.show();
    }
    public static void main(String[] args) {
        Client client = new Client();
        Parent arg = new Child();              //针对父类编程,声明父类,实例化子类
        //Child arg = new Child();             //声明子类,new 子类,易于理解,但是不推荐
        client.call(parent);                   //将 Child 对象传递给函数
    }
}
```

从多态的实现代码中可以知道：

(1) 只有继承，子类才有机会重写父类同样的方法，才能体现对象被替换后，相同接口表现不同的行为。

(2) 子类可以直接重用父类方法，但是那样体现不出执行时行为所体现的不同结果，只有重写了父类的方法，运行时才能体现不同的行为。

(3) 多态的关键点是父子关系中相同行为的不同表现，相同行为体现在函数的签名一致，即客户看到的对象服务接口完全一致。

(4) 另一方面是编写代码时，针对父类编程，只有这样才能够利用向上转型，将所有的子类对象一般化对待，代码才具有一般性，才能达到同样的代码表现出不同的行为特点。

```
public void call(Parent arg) {                 //针对父类编程
    arg.show();
}
```

call 函数参数类型为 Parent，它是基类，是所有 Parent 子类的一般化类型。call 函数接收

不同的 Parent 子类，所表现的行为也不同。

> 构造函数是不能被继承的，只能在子类初始化时调用，所以不存在重写问题，构造函数只能在一个类中被重载。

5.10 微妙的状态多态

例 5-36 一个简单的 Rectangle 类型。

```
//Rectangle.java
package hust.poly;

public class Rectangle {
    private double height;
    private double width;

    public double getHeight() {
        return height;
    }

    public void setHeight(double height) {
        this.height = height;
    }

    public double getWidth() {
        return width;
    }

    public void setWidth(double width) {
        this.width = width;
    }
    //其他辅助功能
    public void other(Rectangle rect) {
        rect.setWidth(30);
        rect.setHeight(40);
        System.out.println(area());
    }
    //计算面积
    public double area() {
        return height * width;
    }
}
```

这个简单的 Rectangle 类型通常情况下运行得很好，只有针对属性的读写函数。但是可能根据需要产生一个正方形。一般来说，正方形确实是一个特殊的矩形，是可以从 Rectangle 继承的，如图 5-18 所示。

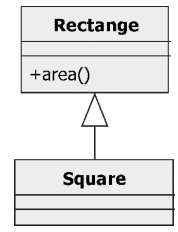

图 5-18 继承 Rectangle 形成的 Square 类型

但是正方形是不需要同时具有 height 和 width 两个属性的，它的四边相等，只需要一个属性即可，但是由于它从 Rectangle 继承，所以也就同时具有了这两个属性，这就产生了微妙的问题。

正方形继承了 setHeight 和 setWidth 函数，为了使它们在调用时保证四边相等，需要重写这两个函数。

例 5-37 通过继承 Rectangle 实现的 Square 类型。

```
//Square.java
public class Square extends Rectangle {
    public void setHeight(double height) {
        super.setHeight(height);
        super.setWidth(height);
    }

    public void setWidth(double width) {
        super.setWidth(width);
        super.setHeight(width);
    }
}
```

通过重写之后，保证了正方形的边长每次都是相等的。

```
public static void main(String[] args) {
    Square square = new Square();
    square.setHeight(10);                    //边长为 10 的正方形
    square.setWidth(20);                     //边长为 20 的正方形
}
```

现在，错误即将产生了，在 Rectangle 类中存在一个针对 Rectangle 编写的函数。

```
public void other (Rectangle rect) {
    rect.setWidth(30);
    rect.setHeight(40);
    System.out.println(area());              //微妙的错误!
}
```

此函数的功能就是根据需要将矩形的宽更改为 30，高更改为 40，合理的面积输出应该是 120，这是很正常的代码，没有任何问题。但当传递 Square 进入函数时将会产生微妙的问题。打印的结果竟然是 160，对于编写 Rectangle 的人，本意并不是这样的，他可能也不会预料到 other 函数竟然能输出 160。

```
public static void main(String[] args) {
    Square square = new Square();
    Rectangle rect = new Rectangle();
    rect.other(square);
}
```

传递 square 之后，other 函数 rect.setHeight(40)语句对于正方形将所有边长更改为 40，这个 rect.setWidth(30)语句没有起到任何作用。

由于 Rectangle 对象的 height 和 width 是可以独立变化的，而 Square 的边并不是独立变化的，导致 Square 重写父类的边长属性设置函数，问题才出现的，所以 Square 从 Rectangle 继承并不恰当，尤其是改变了 Rectangle 类型的 height 和 width 独立变化的特性。

> 多态是行为上的多种形态，而不是状态上的多种形态。

由于 Square 在属性上改变了 Rectangle 的约定，这种重写导致的多态性是危险了，不提倡的，它会导致很多细节上的、不易发现的问题，重写父类函数，主要是针对父类的行为，而不是针对属性，行为上的多态性才是值得关注的。

5.11　深入多态

类的继承关系如图 5-19 所示。

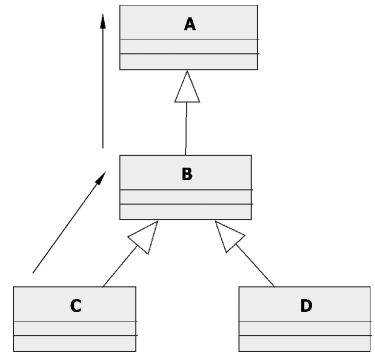

图 5-19　类的层次关系

例 5-38　一个简单而又详细的阐述多态的示例。

```
//PolymorphismTest.java
package hust.poly;

class A {
    public String f(D obj) {return ("A and D");}
    public String f(A obj) {return ("A and A");}
}

class B extends A {
    public String f(B obj) {return ("B and B");}
    public String f(A obj) {return ("B and A");}
}

class C extends B{}
class D extends B{}

public class PolymorphismTest {
    public static void main(String[] args) {
        A a1 = new A();
        A a2 = new B();
        B b = new B();
        C c = new C();
        D d = new D();
        System.out.println(a1.f(b));        //A and A
        System.out.println(a1.f(c));        //A and A
        System.out.println(a1.f(d));        //A and D
        System.out.println(a2.f(b));        //B and A
        System.out.println(a2.f(c));        //B and A
        System.out.println(a2.f(d));        //A and D
        System.out.println(b.f(b));         //B and B
        System.out.println(b.f(c));         //B and B
        System.out.println(b.f(d));         //A and D
    }
}
```

运行结果：

```
A and A
A and A
A and D
B and A
B and A
A and D
B and B
```

```
B and B
A and D
```

这个例子比较全面地讲述了产生多态的多种情形。

(1) System.out.println(a1.f(b))；运行结果：A and A。分析过程如表 5-1 所示。

表 5-1 a1.f(b)的调用过程

代 码	解 释
A a1=new A()；	a1 是 A 对象
B b=new B()；	b 是 B 对象
a1.f(b)	a1 中只有函数 f(D)和 f(A)
	b 作为参数不匹配任何一个函数，自动向上转型为 A，匹配 f(A)函数

说明：b 向上只有 A 类型可以转换。

(2) System.out.println(a1.f(c))；运行结果：A and A。分析过程如表 5-2 所示。

表 5-2 a1.f(c)的调用过程

代 码	解 释
A a1=new A()；	a1 是 A 对象
C c=new C()；	c 是 C 对象
a1.f(c)	a1 中只有函数 f(D)和 f(A)
	c 作为参数不匹配任何一个函数，自动向上转型为 A，匹配 f(A)函数

说明：c 向上可以转换为 B 和 A 类型，转换到 B 时，依然找不到匹配的函数，继续向上转型到 A。

(3) System.out.println(a1.f(d))；运行结果：A and D。分析过程如表 5-3 所示。

表 5-3 a1.f(d)的调用过程

代 码	解 释
A a1=new A()；	a1 是 A 对象
D d=new D()；	d 是 D 对象
a1.f(d)	a1 中只有函数 f(D)和 f(A)
	d 作为参数可以匹配到函数 f(D)

(4) System.out.println(a2.f(b))；运行结果：B and A。分析过程如表 5-4 所示。

表 5-4 a2.f(b)的调用过程

代 码	解 释
A a2=new B()；	a2 是 B 对象，但是被当做 A 来操作，内存中实际存在的是 B 对象
B b=new B()；	b 是 B 对象
a2.f(b)	a2 中函数有
	A 类中的 f(D)、f(A)
	B 类中的 f(B)、f(A)(f(A)被 B 重写)
	a2 在使用时只能使用 A 类中的函数
	b 作为参数匹配 A 中函数，只能匹配到 f(A)，由于 f(A)被 B 重写了，所以执行的是 B 的 f(A)

(5) System.out.println(a2.f(c))；运行结果：B and A。分析过程如表 5-5 所示。

表 5-5　a2.f(c)的调用过程

代　　码	解　　释
A a2＝new B()；	a2 是 B 对象，但是被当做 A 来操作，内存中实际存在的是 B 对象
C c＝new C()；	c 是 C 对象
a2.f(c)	a2 中函数有 A 类中的 f(D)、f(A) B 类中的 f(B)、f(A)(f(A)被 B 重写) a2 在使用时只能使用 A 类中的函数 c 作为参数匹配 A 中函数，只能匹配到 f(A)，由于 f(A)被 B 重写了，所以执行的是 B 的 f(A)

说明：c 的直接父类是 B，但是 c 在匹配执行函数时，不能直接匹配 B 对象中的 f(B)，因为通过 a2 调用，只能使用 A 中的函数，而看不到 B 中的函数，即使内存中的实际对象是 B 也不行。所以 c 直接向上转型到 A，匹配 A 中的 f(A)函数。

(6) System.out.println(a2.f(d))；运行结果：A and D。分析过程如表 5-6 所示。

表 5-6　a2.f(d)的调用过程

代　　码	解　　释
A a2＝new B()；	a2 是 B 对象，但是被当做 A 来操作，内存中实际存在的是 B 对象
D d＝new D()；	d 是 D 对象
a2.f(d)	a2 中函数有 A 类中的 f(D)、f(A) B 类中的 f(B)、f(A)(f(A)被 B 重写) a2 在使用时只能使用 A 类中的函数 d 作为参数可以匹配到 A 中 f(D)

(7) System.out.println(b.f(b))；运行结果：B and B。分析过程如表 5-7 所示。

表 5-7　b.f(b)的调用过程

代　　码	解　　释
B b＝new B()；	b 是 B 对象
b.f(b)	b 中的函数有 f(D)(从 A 继承的)、f(B)、f(A)(f(A)被 B 重写) b 作为参数匹配到 B 中的函数 f(B)

(8) System.out.println(b.f(c))；运行结果：B and B。分析过程如表 5-8 所示。

表 5-8　b.f(c)的调用过程

代　　码	解　　释
B b＝new B()；	b 是 B 对象
C c＝new C()；	c 是 C 对象
b.f(c)	b 中的函数有 f(D)(从 A 继承的)、f(B)、f(A)(f(A)被 B 重写) c 作为参数向上转型为 B，可以匹配 f(B)函数

(9) System.out.println(b.f(d))；运行结果：A and D。分析过程如表 5-9 所示。

表 5-9 b.f(d)的调用过程

代 码	解 释
B b=new B()；	b 是 B 对象
D d=new D()；	d 是 D 对象
b.f(d)	b 中的函数有 f(D)(从 A 继承的)、f(B)、f(A)(f(A)被 B 重写) d 作为参数匹配 f(D)函数

充分理解这个例子和构成多态的步骤，就可以基本理解多态机制了，要想充分理解和应用，可以继续学习一些经典设计模式，能够达到熟练应用的程度。

5.12 多态的形象比喻

面向父类编程中，父类就像一个面具(见图 5-20)。

客户只能通过面具和后面的真正人物交流，面具遮挡住的部分客户无法看见和交流，面具就相当于声明的父类。后面的真正实体就是子类，子类再多的个性化特征也都被面具所遮挡，导致无法交互。

图 5-20 面具

```
Parent obj = new Child();                    //父类遮挡住了子类所有的个性化行为
```

而真正执行的行为是后面的子类对象，因为它才是在内存中真正存在的实体对象。无论前面带了多少个面具，万变不离其宗。

图 5-21 表示了一个典型的继承关系，图 5-22 形象地模拟了这个关系中的类和其中的函数，继承关系如 UML 图，A 类具有函数 f(D)、f(A)。B 类具有自有函数 f(B)以及从 A 继承过来的 f(D)，同时重写了 A 的 f(A)函数。C 继承 B，完全继承了 B 的函数 f(D)、f(A)、f(B)。

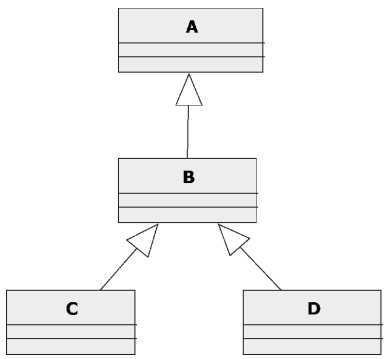

图 5-21 一个典型的类继承关系

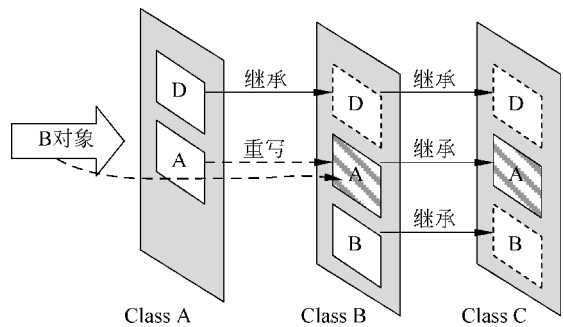

图 5-22 模拟多态函数的调用过程

下面一段代码：

```
B b = new B();
A a = new B();
a.f(b);
```

此时如果一个 B 对象作为参数从面向父类 A 编程的代码中穿过，B 无法直接匹配 f(A)、

f(D),b 只能转型为 A 进行穿越,到达 B 类的 f(A),由于 f(A)被 B 类重写,已经不是继承 A 类的 f(A)函数了,所以系统将会执行 B 类重写的 f(A)函数。

可以看出,如果 B 作为参数从 A 类穿过,永远都无法使用 B 类中很匹配的 f(B)函数,因为它根本就无法到达。面具 A 类阻挡了 B 类和 C 类中除 A 类函数之外的一切。

理解到这点,面向对象中的多态技术就不难理解和使用了。

5.13 子类可以更具体

重写父类函数时,通常要求子类重写的函数要和父类一致。但从图 5-23 中会发现,也可以不一致,Child 重写父类的 handle 函数时,返回值和父类并不一致。那么这种情况还能称为重写吗?它能产生多态吗?

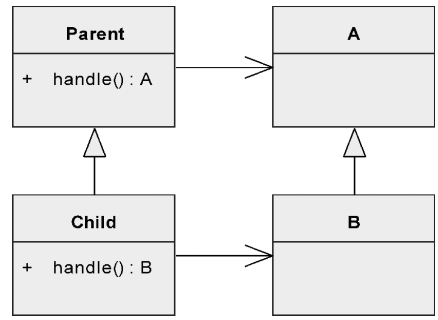

图 5-23 Child 重写父类的 handle 函数

例 5-39 重写父类函数的返回值可以比父类的返回值更具体。

```
//Parent.java
package hust.poly;

public class Parent {
    public A handle() {
        System.out.println("parent");
        return new B();
    }

    public static void main(String[] args) {
        Parent p = new Parent();                //1
        A a = p.handle();                       //2

        p = new Child();                        //3
        a = p.handle();                         //4

        Child child = new Child();              //5
        a = child.handle();                     //6
        B b = child.handle();                   //7

    }
}

class Child extends Parent {
    public B handle() {
        System.out.println("child");
        return new B();
    }
}

class A {}
class B extends A {}
```

运行结果:

```
parent
child
child
child
```

这种形式的代码依然可以称为子类 Child 重写了父类 Parent 的函数 handle。子类重写父类函数时，可以返回比父类更具体的类型，这是没有问题的。因为返回的类型 B 可以完全被当做父类 A，不违反针对父类编程的向上类型转换，另外根据函数签名规则，返回值也不是签名的一部分。

从 main 函数中可以看出，使用方法都是正确的，子类的 handle 函数返回值完全不影响代码的使用，看着和返回 A 没什么区别。

第 1 行创建了一个 Parent 对象，它的 handle 调用返回的自然是 A 类型。

第 3 行定义了一个子类 Child，但是被当做父类 Parent 来看待，所以它返回的值依然是 A（注意这里不能声明 B 变量，因为 Parent 中没有返回 B 的函数存在），即使实际上返回的是 B 对象，但是被当做 A 毫无问题。

第 5 行定义的就是一个 Child 对象，也是被当做 Child 来使用的，所以它的函数 handle 返回的 B 类型既可以被当做 A 也可以被当做 B 来看待。第 6 行和第 7 行没有任何问题。

但是这种写法在一个类中，进行重载设计将是错误的，它违反了在一个类中不能有重复函数的约定。

所以，重写函数时，可以更改函数返回值（只能是更具体的类型），但是重载时不能通过修改函数返回值达到创建新的重载函数的目的。

思考：

（1）为什么在重写时，不能将返回值改写成比父类更一般化（泛化）的类型呢？

（2）子类在重写父类函数时，是否可以将函数的访问权限放大呢？为什么？如下代码将 protected 重写时变为 public：

```
public class Parent {
    protected A handle() {
        System.out.println("parent");
        return new B();
    }
}
public class Child extends Parent {
    public A handle() {
        System.out.println("child");
        return new B();
    }
}
```

（3）子类重写父类函数时，可以将函数参数更具体些吗？例如 A 中的 handle(A a)，B 再重写 handle 时将其具体成这样：handle(B b)，可以吗？

5.14 抽象类

在面向父类编程的过程中，抽象出来的父类具有一般化特质，很多时候，这样的父类函数只是一个抽象化的概念，它们被定义出来只是为了在面向父类编程时统一接口服务。

如图 5-24 所示，定义一个人类，它具有说话功能。但是人类此时仅仅是一个概念，其中的说话功能是所有人类共同具有的行为，人这个抽象的概念中，说话功能并不能确切地被定义出来，只有具体到了某个人身上才能被实现。

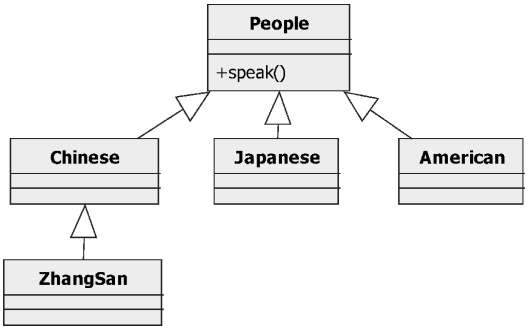

图 5-24　人的继承关系

在中国人身上可以被定义出来具体的说话能力，但是也只能定义到说中国话，如果到了具体的一个中国人，那么说话的功能将会被更明确地实现。

所以，父类在一般化子类时，为了统一接口的目的，有时会定义出一些无法实现的行为。

```
public void call(People people) {
    people.speak();
}
```

此函数面向父类 People 设计，它适合所有的人类使用，但是此时 People 的子类没有被定义，People 中的 speak 也无法具体实现。此段代码是一段通用的代码，不能等待所有子类设计完成之后再设计 call 函数，这种等待也不现实，这种期望也不可能实现。所以系统需要提供一种机制来定义一种不用实现的函数，这种函数称为抽象函数。

```
抽象类与抽象函数：
    public abstract class People {      //抽象类
        public abstract void speak();//抽象函数
    }
```

抽象函数是一种没有实现体的函数，它使用 abstract 关键字修饰，它表达了一种不完整的概念，类中如果存在抽象的、没有被实现的函数，那么该类也是一种不完整的类型，也是一种抽象的类型，称为抽象类。

抽象类由于代码不完整，等待子类去实现，所以不能直接使用。

```
People people = new People();                    //错误
```

这样定义的 people 没有意义，因为其中的很多函数没有实现，无法调用，所以系统在编译期就会发现这样的错误，不允许创建抽象类型对象。

只有在子类实现了其中的抽象函数后，构成了一个完整的、具体的类型之后，才能被使用。

```
public class Chinese extends People {
    public void speak() {
        System.out.println("中国话");
```

```
    }

    public static void main(String[] args) {
        People people = new Chinese();
        people.speak();
    }
}
```

子类继承抽象的父类之后，必须实现父类中抽象的方法，以便使自己成为一个完整的、具体的类型。当然，子类也可以出于程序需要，不去实现父类的抽象方法，那么，这样的子类依然是不完整的，也是抽象的。

例 5-40 没完全实现抽象父类中的抽象方法，子类依然是一个抽象类型。

```
public abstract class Chinese extends People {
    //没有实现 speak 函数，添加了一个新的函数

    public void cook() {
        //具体的 cook 代码
    }
}
```

没有完全实现父类抽象函数的子类，也可以添加自己的一些函数，例如 cook，它是一个具体的函数，但是由于没有完成父类的抽象函数，它依然是一个抽象类，不能被直接使用。

> 一个抽象类中，至少有一个以上的抽象函数，其中可以混合定义具体的函数。
>
> 一个类中如果没有任何抽象元素，它也可以被定义为一个抽象类型。这表示设计者不希望用户直接实例化它，只希望通过子类来实现具体的应用。

抽象类更像一个填空题，缺少的部分等待子类填写完整，之后，它就会变成一个完整、通顺的语句。也可以把它理解成为一个疑问句，解决了其中的疑问，那么它将会变成一个没有疑问的陈述句。

对于父类不能完全确定的实现函数，是否可以使用空实现体呢？

```
public class People {
    public void speak() {
        //什么也不写的空实现体
    }
}
```

这样的函数虽然没有任何具体的实现，可它也是一个具体的函数，这样的类也是一个具体的类。这样编写的 People 可以直接使用，客户也可以调用 speak 函数，但是执行结果却什么也没有，客户可能会觉得迷惑，因为它没有任何反应，所以不建议这样做。

类行为是向外界提供服务的，既然没有确切的服务，那就应该等到有了具体的服务之后，再提供给客户。

如果出于某种需要，确实需要定义具体的类型，但是其中的有些方法确实无法实现，可以类似例 5-41 这样处理。

例 5-41 使用抛出异常来表示此方法没有实现。

```
public class People {
```

```
    public void speak() throws IllegalAccessException {
        throw new IllegalAccessException("没有实现,不可访问");
    }
}
```

当用户调用此函数时,系统会抛出一个异常,这样做虽然不太合适,但是,至少客户知道此函数没有实现,不能直接调用,从而改用其他方式,这样要比一个空实现更友好。

抽象类(Abstract Class)较普通的具体类(Concrete Class)实际上并无太大区别,只是它是一个具有抽象方法的不能直接实例化的类,其他和普通类型没有区别。

所以:

(1) 抽象类可以有构造方法。
(2) 抽象类中可以有普通成员变量。
(3) 抽象类中可以包含非抽象的普通方法。
(4) 抽象类中的抽象方法的访问类型可以是 public、protected 和默认类型。
(5) 抽象类中可以包含静态方法。
(6) 抽象类中可以包含静态成员变量,抽象类中的静态成员变量的访问类型可以任意。
(7) 抽象类只能继承一个类。

使用代码描述一个简单的树状结构,如图 5-25 所示。

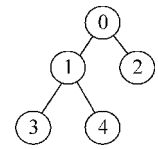

图 5-25 一个简单的树状结构

例 5-42 使用抽象类定义一个树的节点。

```
//Node.java
package hust.node;
import java.util.ArrayList;
import java.util.List;

public abstract class Node {
    protected String name;
    protected int data;
    public abstract Node add(Node child);
}

class ConcreteNode extends Node {
    private List<Node> children = new ArrayList<Node>();

    public ConcreteNode(String name,int data) {
        this.name = name;
        this.data = data;
    }

    public Node add(Node child) {
        children.add(child);
        return this;
    }

    public static void main(String[] args) {
        Node root = new ConcreteNode("root",0);
        Node first = new ConcreteNode("first",1);
        root.add(first).add(new ConcreteNode("second",2));
```

```
            first.add(new ConcreteNode("third",3)).add(new ConcreteNode("fourth",4));
        }
    }
```

执行之后 root 根节点的数据结构如图 5-26 所示。与图例中的树状结构一致。

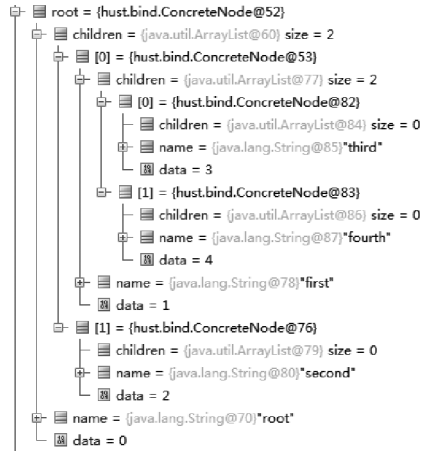

图 5-26　root 根节点的数据结构

Node 类型作为抽象类，它定义了所有节点的共同特征：name、data 以及可以添加子节点 add 的功能。

ConcreteNode 是一个具体的节点，它实现了抽象类 Node 的要求，这样客户可以使用 ConcreteNode 进行树的创建工作。

这里的叶节点没有做特殊限制。节点 3、4 作为叶节点，应该不允许添加子节点。但是从程序来看依然可以添加子节点，如何做限制呢？

例 5-43　使用抛出异常的方法限制叶节点不可以添加子节点。

```
//Node.java
package hust.node;
import java.util.ArrayList;
import java.util.List;

public abstract class Node {
    protected String name;
    protected int data;
    //定义了可能出现的错误
    public abstract Node add(Node child) throws IllegalAccessException;
}

class ConcreteNode extends Node {
    private List<Node> children = new ArrayList<Node>();

    public ConcreteNode(String name,int data) {
        this.name = name;
        this.data = data;
    }

    public Node add(Node child) throws IllegalAccessException {
        children.add(child);
        return this;
    }

    public static void main(String[] args) throws IllegalAccessException {
        Node root = new ConcreteNode("root",0);
        Node first = new ConcreteNode("first",1);
        root.add(first).add(new ConcreteNode("second",2));
        Node leaf3 = new LeafNode("thrid",3);
        Node leaf4 = new LeafNode("fourth",4);
        first.add(leaf3).add(leaf4);
        leaf3.add(new ConcreteNode("fifth",5)); //叶节点添加子节点出现错误
    }
}
//叶节点定义
```

```
class LeafNode extends ConcreteNode {
    public LeafNode(String name,int data) {
        super(name,data);
    }

    public Node add(Node child) throws IllegalAccessException {
        throw new IllegalAccessException("叶节点不可以添加子节点!");
    }
}
```

针对叶节点的特殊性，设计了 LeafNode 类型，它也是一个 Node，但是它不具备添加子节点的功能，所以在实现 add 函数时，抛出了一个异常通知用户，此节点 add 行为不可使用。如果客户能够更改 Node 代码，在 Node 类中添加一个针对叶节点的 add 方法。子节点可以不用继承于 Node 类，可以单独设计一个 LeafNode，避免实现无用的 add 函数。

```
public abstract Node add(LeafNode leaf);
```

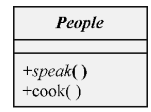

图 5-27　抽象类的 UML 图

抽象类 UML 图 5-27 类名用斜体标识，其中的抽象函数使用斜体。

从多态性机制上可以看出，针对父类编写的代码将会阻挡所有子类特有函数的调用，所以在定义通用的父类时，需要把所有共有的特性全部定义出来，即使当时不能马上实现的行为也要定义出来，否则面向父类编程的过程中将无法使用子类的这些函数，这就是设计抽象类的目的。

5.15　接口

> 接口可以理解为完全抽象的类，但是比抽象类更进一步，它不是类，而是一个特殊的名称：接口，使用 interface 关键字标识。

Java 只允许单根继承，也就是只能有一个父类，但是接口却不同，子类可以实现多个接口，如图 5-28 所示。

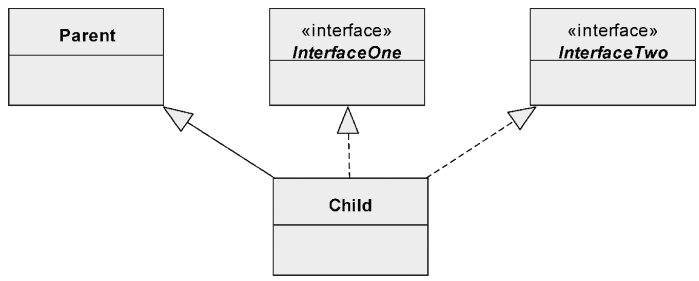

图 5-28　接口的多重实现

接口实现

```
public class Child extends Parent implements InterfaceOne,InterfaceTwo {
}
```

接口只提供了对象的所有行为签名，而没有任何实现。

它不可以直接被实例化，它只能在被子类实现为一个具体类型后才能使用。

```
InterfaceOne one = new Child();
InterfaceTwo two = new Child();
```

实例化具体的 Child 子类才可以，否则这样是错误的：

```
InterfaceOne one = new InterfaceOne();
```

因为接口没有任何实现，没有任何现实意义，完全抽象的、不具体的、直接实例化它没有意义。

接口可以理解成一个非常纯净的类，它没有任何杂质（具体实现），任何类实现接口中的函数都是完全自主的，没有任何干扰的，因为子类从接口没有继承任何现有的东西，而仅仅是一些行为规范。但使用接口的同时，又带来面向父类编程的优势，代码更加通用、清晰。

interface 允许人们通过创建一个能够被向上转型为多种基类的类型来实现某种类似多重继承的特性。

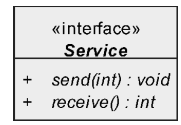

图 5-29 接口的 UML 图

5.15.1 接口的定义

接口的 UML 图表示如图 5-29 所示。

接口定义

```
//Service.java
package hust.intf;

public interface Service {
    int data = 10;
    void send(int data);
    int receive();
}
```

接口定义只是将原来定义类的 class 关键字换成 interface 关键字。

接口中的方法定义和通常的方法定义一样，只是没有实现体。接口中的方法只能是公有的 public abstract，所以可以省略不写。接口中的 data 变量看似是一个成员变量，但实际上默认它是一个公有的静态的 final 变量（也就是一个常量），也是被省略了。完整的 Service 类应该是这样：

```
public interface Service {
    public static final int data = 10;
    public abstract void send(int data);
    public abstract int receive();
}
```

接口是统一的服务规范，客户都是可以全部使用的。

5.15.2 接口的实现

接口是完全抽象的类型，只有实现成为一个具体类型才能被客户实例化。

接口的实现 UML 图如图 5-30 所示。

例 5-44　实现 Service 接口。

```
public class ConcreteService implements Service {
    public void send(int data) {
        System.out.println(data);
    }

    public int receive() {
        return 10;
    }
}
```

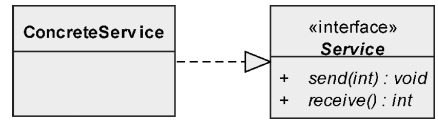

图 5-30　接口的实现

使用关键字 implements 后面加上需要实现的接口名称即可，如果实现多个接口，用逗号分隔。

实现接口的类要求必须实现接口所要求的所有函数，才能构成一个具体的类型。

> 完全实现了所有接口方法的类就是一个具体的类。
> 部分实现了接口方法的类是一个抽象的类。
> 接口通过实现(implements)演变成为一个类(class)。
> 接口可以通过继承(extends)创建新的子接口。

如果未完全实现接口要求的所有函数，那么它的实现类将是一个抽象类，如图 5-31 所示。

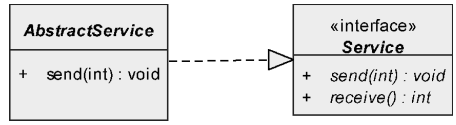

图 5-31　未完全实现接口函数的抽象类

例 5-45　未完全实现接口函数的类依然是抽象类。

```
public abstract class AbstractService implements Service {
    public void send(int data) {
        System.out.println(data);
    }
}
```

可以把接口理解为一个完全抽象的类，只是它的用法稍有区别。

> (1) 它使用关键字 implements 实现到类的演变。
> (2) 可以多重实现(类似多重继承，但是更纯净!)。
> (3) 方法默认全部公有 public。
> (4) 所有方法只有定义，没有实现。
> (5) 所有变量默认都是公有静态 final 的常量，而且必须被显式初始化。

其他都与普通类使用方法没有任何区别。

例 5-46　接口的实现与应用。

```
//Service.java
package hust.intf;
```

```
public interface Service {
    void send(int data);
    int receive();
}

abstract class AbstractService implements Service {
    public void send(int data) {
        System.out.println(data);
    }
}

class ConcreteService extends AbstractService {
    public int receive() {
        return 10;
    }

    public static void main(String[] args) {
        Service service = new ConcreteService();
        service.send(10);
    }

}
```

可以看出,接口的使用和普通的类没有任何区别。

5.15.3 接口的多重实现

接口可以多重实现,如图 5-32 所示,类似多重继承。

例 5-47 接口的多重实现。

```
//Service.java
package hust.intf;

public interface Service {
    void send(int data);
    int receive();
}

interface OtherService {
    void send(int data);
}

class ConcreteService implements Service,OtherService {
    public void send(int data) {
        System.out.println(data);
    }

    public int receive() {
        return 10;
    }
}
```

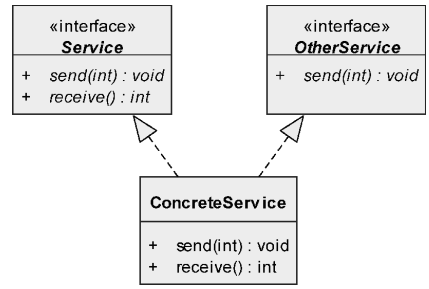

图 5-32 接口的多重实现

ConcreteService 实现了两个接口，但是这两个接口中都包含有 send 函数，而且它们的签名相同，在实现了两个接口时都需要实现这个函数，它们将合并成为一个实现。

ConcreteService 向上转型，既可以被认为是 Service，也可以被认为是 OtherService。

```
Service service = new ConcreteService();
OtherService otherService = new ConcreteService();
```

这种能够向上转型为多个类型的特点很有用处。

5.15.4　一个隐含的问题

当多重实现多个接口时，可能会出现被实现函数混淆问题，如图 5-33 所示，Service 和 OtherService 中都有 send 函数，但是它们定义得并不相同，这就导致实现类 ConcreteService 在实现时存在问题，同时实现不同的 send，会导致在同一个类中出现同样的两个 send 函数（只是返回值不同），这是类规则不允许的，如果只实现一个接口中的 send 函数而另一个不去实现，这又不符合接口实现要求，所以此时问题出现了。

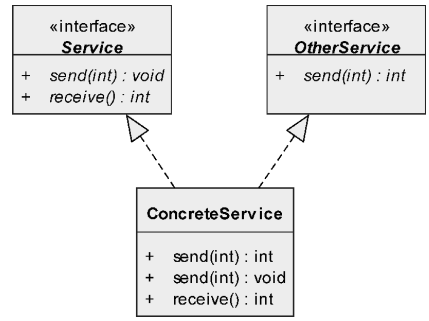

图 5-33　多重接口实现的函数问题

例 5-48　在多重实现接口时出现函数冲突问题。

```
//Service.java
package hust.intf;

public interface Service {
    void send(int data);
    int receive();
}

interface OtherService {
    int send(int data);                    //与 Service 中的 send 函数相似，只是函数返回值部分不同
}

class ConcreteService implements Service,OtherService {
    public void send(int data) {
        System.out.println(data);
    }

    public int send(int data) {            //与上一个 send 函数在定义上冲突
        System.out.println(data);
        return 0;
    }

    public int receive() {
        return 10;
    }
}
```

在实现两个接口时，如果出现函数名称冲突的问题，问题将无法解决，只能尽可能避免此问题的出现。

例 5-49 出现了函数冲突，但是返回值更具体，可以自动解决。

```
//Client.java
package hust.intf;

public class Client {
    void test(DevicedGetter d) {
       Deviced d2 = d.get();
    }
}

class Base {}
class Deviced extends Base {}

interface OrdinaryGetter {
    Base get();
}
interface DevicedGetter extends OrdinaryGetter {
    Deviced get();
}
```

子接口 DevicedGetter 继承了 OrdinaryGetter 接口，重写了 get 函数，但是返回值是 Base 的子类型，上述代码说明了子类是可以返回比父类更具体的类型的。虽然在一个类中这样两个函数是不允许同时存在的，但是重写确实是可行的。

思考：图 5-33 中，两个接口的 send 函数的返回值如果有继承关系，此时的多重实现是否可行？

5.15.5 接口的继承

接口继承

```
interface ChildService extends Service {
}
```

和类的继承类似，但是接口的继承却可以多重继承，之所以这样规定，是因为接口相比普通类而言，更加纯净，多重继承不会带来潜在的干扰，如图 5-34 所示。

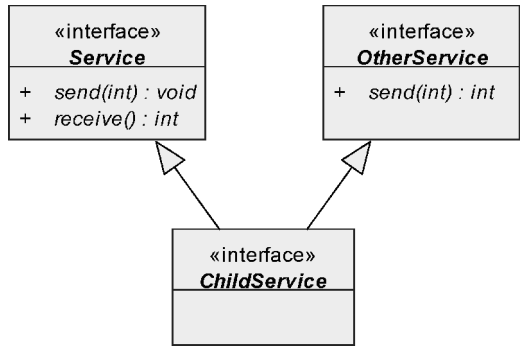

图 5-34 接口的多重继承

接口的多重继承

```
interface ChildService extends Service,OtherServie {
}
```

多重继承后，子接口拥有了父类所有的接口定义，它们依然只是定义，没有任何实现。

在接口的多重继承上同样可能存在函数名冲突的问题，也就是不同返回值的相同函数导致子类中出现了两个不符合要求的重复函数定义。

上面的 UML 图 5-34 中就存在这样的问题，send 函数在单独的 Service 和 OtherService 接口中不存在任何问题，但是当它们共同作为 ChildService 的父类时，导致子类同时拥有两个 send 函数，它们无法同时在子类中存在，导致问题出现，这个问题在不修改父接口的情况下无法解决。

换一个思路：

```
interface ChildService extends Service {
    OtherServie getOtherService();
}
```

这样也可以达到目的。将另一个接口通过一个函数 getOtherServicve 返回它的实例，也可以达到类似多重继承接口的目的。

多重继承的目的是在适当的时候可以既被当做 Service，也可以被当做 OtherService 来看待。在直接多重继承冲突的情况下，通过第二种方案一样可以达到要求：它在适当的时候可以被当做 Service 看待，也可以通过调用 getOtherService 的方式返回 OtherService，这样它也可以被当做 OtherService 来看待了。

接口和抽象类有很多相似之处，但是：

(1) 接口中不能有构造方法。

(2) 接口中没有普通成员变量，只有常量。

(3) 接口中的所有方法必须都是抽象的，不能有非抽象的普通方法。

(4) 接口中的抽象方法只能是 public 类型的，并且默认即为 public abstract 类型。

(5) 接口中不能包含静态方法。

(6) 抽象类和接口中都可以包含静态成员变量。但接口中定义的变量只能是 public static final 类型，并且默认即为 public static final 类型。

(7) 一个类可以实现多个接口，但只能继承一个类。

5.15.6 适配器模式

如图 5-35 所示，Client 针对抽象类型 Processor 编程，会使 show 函数更加通用，通过传入不同的 Processor 子类，show 函数可以运行实现不同的运行结果。

例 5-50 一个抽象字符串处理器的设计与实现。

```
//Processor.java
package hust.intf;

public abstract class Processor {
```

```
    public abstract String process(String info);
}

class StringProcessor extends Processor {
    public String process(String info) {
        info = info.toUpperCase();
        return info;
    }
}

//Client.java
package hust.intf;

public class Client {
    private Processor processor;

    public Client(Processor processor) {
        this.processor = processor;
    }

    public void show(String info) {
        processor.process(info);
        info = "<<" + info + ">>";
        System.out.println(info);
    }

    public static void main(String[] args) {
        Client client = new Client(new StringProcessor());
        client.show("hello world");
    }
}
```

运行结果：

```
<<HELLO WORLD>>
```

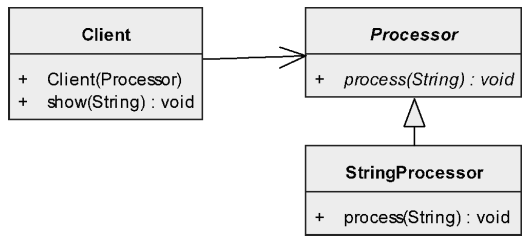

图 5-35　针对抽象类 Processor 编程

Processor 作为处理字符串算法的基类，提供了一个接口 process，规定所有的子类都需要提供具体的处理字符串的算法，Client 类使用 Processor 提供的算法，完成自己的一些逻辑处理，Client 针对 Processor 编写，适合所有基于 Processor 实现的算法，上例中的 StringProcessor 就是 Processor 的一个具体实现。

如果另一个类库中有一个好的字符串处理算法已经写好，它的功能是给出的语句首字母

大写，如图 5-36 中的 OtherHandle 中就含有一个已经写好的 handle 处理函数。

例 5-51　一个第三方已经实现的字符串处理类。

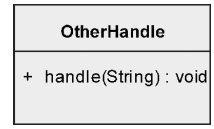

图 5-36　已经存在的 OtherHandle 处理类

```
//OtherHandle.java
package hust.intf;
public class OtherHandle {
    public String handle(String info) {
        String[] words = info.split(" ");
        StringBuffer sb = new StringBuffer();
        for(String word:words) {
            if(word.length() == 1)
                sb.append(word.substring(0,1).toUpperCase());
            if(word.length()> 1)
                sb.append(word.substring(1));
            sb.append(" ");
        }
        return sb.toString();
    }
}
```

OtherHandle 中的 handle 是一个复杂的处理字符串的算法，如果想在 Client 中重用这个 OtherHandle 的功能该如何做呢？由于 Client 只接收 Processor 类型，严重依赖 Processor 类，无法支持 OtherHandle，非常遗憾。但如果将 Processor 设计成为接口，曙光将出现。

修改的前提是：

(1) OtherHandle 类在第三方类库中，开发者无法修改。

(2) 开发者可以修改 Processor 类。

开发过程中，重用第三方类库功能的情况经常出现，如果被限制，或者大幅度更改代码，开发效率将大大降低。

重用第三方 OtherHandle 的方法如图 5-37 所示。

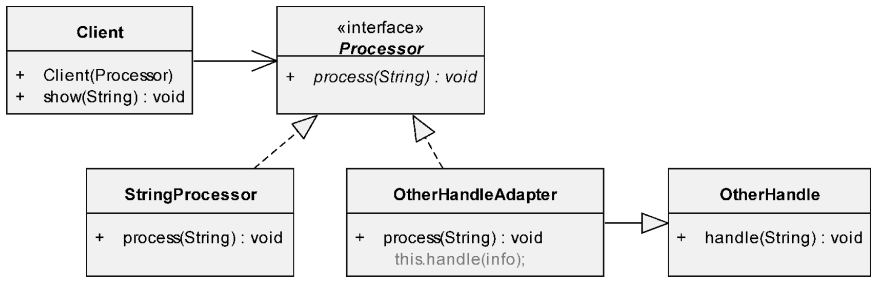

图 5-37　使用策略模式重用 OtherHandle

例 5-52　使用策略模式实现的重用第三方字符串处理类的方法。

```
//Processor.java
package hust.intf;

public interface Processor {
    String process(String info);
}
```

```
//StringProcessor.java
package hust.intf;

public class StringProcessor implements Processor {
    public String process(String info) {
        info = info.toUpperCase();
        return info;
    }
}

//Client.java
package hust.intf;
public class Client {
    private Processor processor;

    public Client(Processor processor) {
        this.processor = processor;
    }

    public void show(String info) {
        info = processor.process(info);
        info = "<<" + info + ">>";
        System.out.println(info);
    }

    public static void main(String[] args) {
        Client client = new Client(new StringProcessor());
        client.show("hello world");
        Client otherClient = new Client(new OtherHandleApapter());
        otherClient.show("hello world");
    }
}

//OtherHandle.java
package hust.intf;
public class OtherHandle {
    public String handle(String info) {
        String[] words = info.split(" ");
        StringBuffer sb = new StringBuffer();
        for (String word : words) {
            if (word.length() == 1) ;
            sb.append(word.substring(0,1).toUpperCase());
            if (word.length() > 1)
                sb.append(word.substring(1));
            sb.append(" ");
        }
        return sb.toString();
    }
}

//OtherHandleAdapter.java
package hust.intf;
public class OtherHandleApapter extends OtherHandle implements Processor {
```

```
    public String process(String info) {
        return this.handle(info);
    }
}
```

运行结果：

```
<<HELLO WORLD>>
<<Hello World>>
```

在不更改 OtherHandle 类的情况下，将 Processor 修改为 interface，以及将它的子类继承(extends)改为实现(implements)，这样，阻挡的重用 OtherHandle 算法的障碍基本消除，因为一个类只能有一个基类，但是实现的接口却可以有多个，将 Processor 改为接口，将不会影响原有的继承关系。

新创建一个 OtherHandleAdapter 类型，它继承 OtherHandle，同时实现接口 Processor，由于继承了 OtherHandle 的 handle 函数，子类可以直接使用，在实现接口函数 process 的时候，调用 handle 函数来完成算法，这样就重用了 OtherClient 的代码，原有 Client 也不用更改。此时 OtherHandleAdapter 的功能由 OtherHandle 类提供的函数协助完成，同时它也可以被当做 Processor 使用，它实现了 Processor 要求的接口，一举两得！

由此例子可以看到针对接口的优势，如果 Processor 是一个类，OtherHandleAdapter 将无法再使用继承，因为它已经继承了 OtherHandle，Java 不允许多继承。

上述设计方法实际上就是一种策略模式。

> 这些算法通过继承 Processor 都可以独立变化，算法的变化独立于客户 Client 的使用，在运行期间可以被客户随意替换。

另一种情况：如果开发者无法修改 Processor 类。

也就是不可以将抽象 Processor 改为接口，实际上这种要求就排除了继承 OtherHandle 的可能，因为 Java 只能单根继承，所以只能通过组合重用 OtherHandle 功能。

例 5-53 无法修改 Processor 的情况下，使用适配器模式实现重用第三方函数。

```
//OtherHandleAdapter.java
package hust.intf;
public class OtherHandleApapter extends Processor {
    private OtherHandle otherHandle = new OtherHandle();  //组合使用
    public String process(String info) {
        return otherHandle.handle(info);                  //重用 OtherHandle 代码
    }
    public static void main(String[] args) {
        Client client = new Client(new StringProcessor());
        client.show("hello world");
        Client otherClient = new Client(new OtherHandleApapter());
        otherClient.show("hello world");
    }
}
```

由于 Processor 不能被修改为接口，代码通过继承 Processor 以及组合 OtherHandle 类的方式，重用 OtherHandle 的功能，这样的结果很令人满意，它符合了 Client 中 show 函数参数

的要求：只识别 Processor。

此种设计方法称为适配器设计模式。

> 适配器模式将 OtherHandle 提供的服务轻易地转换到了 Processor 提供的服务，将原本不兼容的接口可以工作在一起。

图 5-38 是适配器设计模式的标准 UML 图。

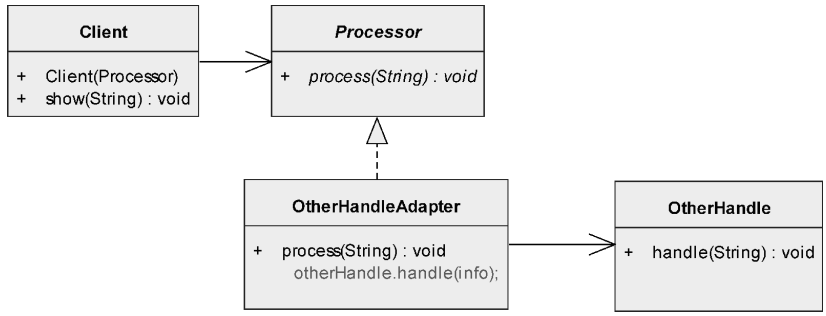

图 5-38 使用适配器模式重用 OtherHandle

通过 UML 了解这个模式的情况一目了然：

(1) Client 只关联 Processor，协助完成自己的逻辑。

(2) OtherHandle 类型不符合 Client 的要求，接口函数也不符合要求。

(3) 通过实现 Processor，形成的 OtherHandleAdapter 类关联了 OtherHandle，在实现接口函数 process 时，通过重用 OtherHandle 现有的功能辅助完成了 process 的逻辑。

OtherHandleAdapter 将原本不兼容的 Processor 和 OtherHandle 类型结合到了一起，使 OtherHandle 也能为 Client 提供服务。

5.15.7 组合设计模式

组合设计模式的 UML 图如图 5-39 如示。

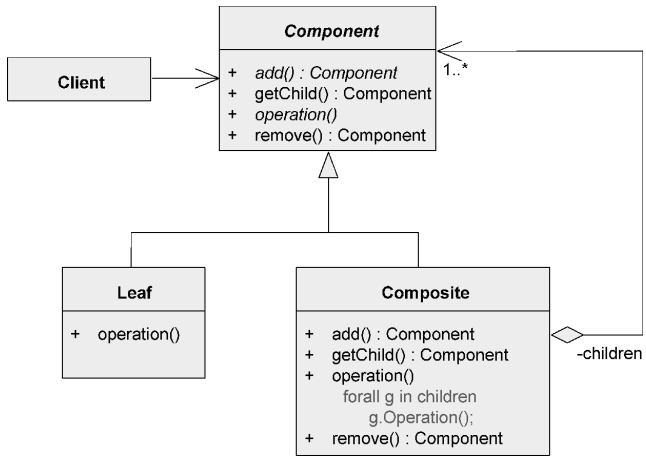

图 5-39 组合设计模式的 UML 图

从图 5-39 中可以看到：

(1) Component 是抽象的。

(2) Composite 继承于 Component，它是具体类，并且它聚合了多个 Component(1..*)，可以知道 Composite 不光充当 Component 的角色，同时也是一个容器，容纳多个 Component。

(3) Component 中含有 add、remove 和 getChild 获得子 Component 的功能，说明它有一个指向与自己同类型 Component 的功能。

(4) 总结以上几点：Component 可以指向自己，Composite 包含多个自己。结论就是 Composite 是一个表达了一种树状结构的图。

> 组合模式示意图：将对象组合成树状结构以表示“部分-整体”的层次结构。Composite 使得用户对单个对象和组合对象的使用具有一致性。

这个模式的关键是 Composite 抽象类，它既可以代表树状结构的节点，又可以代表整个树状结构，这样的特点使得操作整个树状结构时，操作整个树和操作单个节点的方法相同。这样带来的好处就是“部分-整体”的概念消失了，可以通过节点创建任意树状结构，节点就代表树，操作节点就等于在操作树结构，树中包含的节点，实际上就是节点下的子节点，整个树的概念变得没有那么复杂了，都变成了节点操作。

从 UML 中就可以看出：Component 就是树的节点，Composite 继承于 Component，它也是一个节点，但是同时它又充当了一个容器，包含有多个子节点，通过 Composite 操作 Component 实际上就是节点在自己操作自己，因为它们本来就是同一个类型。

组合模式通过定义基本对象和组合对象，可以组合成为更复杂的对象。

通过组合模式用户使用一致的方法操作节点和整个树结构。

用户只针对 Component 编程即可，新节点的创建并不影响客户代码。

组合模式根据具体情况，实现代码和 UML 图稍有不同，代码将具有一般性的 Component 定义为接口，其他类之间的关系和 UML 图一致。

例 5-54 组合设计模式的实现。

```
//Component.java
package hust.intf.composite;
import java.util.List;

public interface Component {
    String getName();
    Component add(Component component);
    Component remove(Component component);
    void operation();
    List<Component> getChild();
    List<Component> getAllChild();
}

//Composite.java
package hust.intf.composite;
import java.util.List;
import java.util.ArrayList;

public class Composite implements Component {
```

```
protected List<Component> children = new ArrayList<Component>();
private String name;

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

public String getName() {
    return name;
}

public Component add(Component component) {
    children.add(component);
    return component;
}

public Component remove(Component component) {
    children.remove(component);
    return component;
}

public void operation() {
    //一些具体的逻辑
    for (int i = 0; i < children.size(); i++) {
        Component component = children.get(i);
        component.operation();
    }
    System.out.println("component:" + name);
}

public List<Component> getChild() {
    return children;
}

//获得所有的子节点
public List<Component> getAllChild() {
    find(this);
    return result;
}

List<Component> result = new ArrayList<Component>();
//递归获得所有的子节点
private void find(Component c) {
    List<Component> ls = c.getChild();
    for (Component o : ls) {
        if (o instanceof Composite)
            find(o);
        result.add(o);
    }
}

public String toString() {
    return name;
```

```
    }

    public int hashCode() {
        return name.hashCode();
    }

    public boolean equals(Object obj) {
        if(obj instanceof Component) {
            Component component = (Component) obj;
            return this.name.equals(component.getName());
        }
        throw new IllegalArgumentException("比较的不是" + this.getClass().getName());
    }
}

//Leaf.java
package hust.intf.composite;
import java.util.List;

public class Leaf implements Component {
    private String name;

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

    public String getName() {
        return name;
    }

    //不支持该函数
    public Component add(Component component) {
        throw new IllegalAccessError("不支持该方法");
    }

    //不支持该函数
    public Component remove(Component component) {
        throw new IllegalAccessError("不支持该方法");
    }

    public void operation() {
        //一些具体的逻辑
        System.out.println("leaf:" + name);
    }

    //没有子节点,返回 null
    public List<Component> getChild() {
        return null;
    }
    //不支持该函数
    public List<Component> getAllChild() {
        return null;
    }
```

```
    public String toString() {
        return name;
    }

    public int hashCode() {
        return name.hashCode();
    }

    public boolean equals(Object obj) {
        if(obj instanceof Component) {
            Component component = (Component) obj;
            return this.name.equals(component.getName());
        }
        throw new IllegalArgumentException("比较的不是" + this.getClass().getName());
    }
}
```

代码实现很简单,只是要注意的是 Leaf 不能有 add 和 remove 方法,因为它是叶节点,不可以再有子节点了,所以将 Leaf 的这两个方法的实现取消了,在用户调用这两个方法时会给出异常。

现在使用 Composite 模式定义一个图 5-40 样式的树状结构。

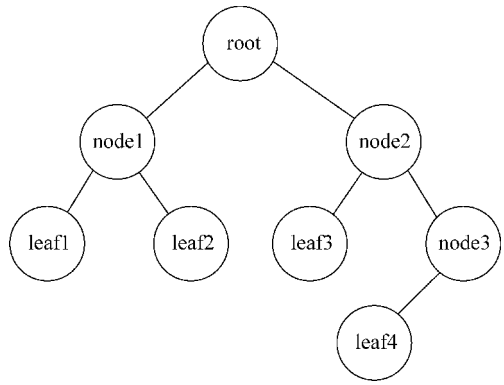

图 5-40　一个典型的树状结构

例 5-55　组合设计模式的典型应用。

```
//Client.java
package hust.intf.composite;
import java.util.List;

public class Client {
    public void handle(Component component) {
        component.operation();
    }

    public static void main(String[] args) {
        //定义根节点
        Component root = new Composite("root");
        Component node1 = new Composite("node1");
        Component leaf1 = new Leaf("leaf1");
```

```
        Component leaf2 = new Leaf("leaf2");
        //定义 node1 节点添加 2 个叶节点
        node1.add(leaf1);
        node1.add(leaf2);
        //将 node1 挂接到 root 节点上
        root.add(node1);

        Component node2 = new Composite("node2");
        Component node3 = new Composite("node3");
        //将 node3 挂接到 node2 节点上
        node2.add(node3);

        Component leaf3 = new Leaf("leaf3");
        Component leaf4 = new Leaf("leaf4");
        //给 node2 挂接 2 个叶节点
        node2.add(leaf3);
        node3.add(leaf4);
        //将 node2 挂接到 root 节点上
        root.add(node2);
        System.out.println("打印根节点下的所有节点");
        List<Component> ls = root.getAllChild();
        for (Component cs : ls) {
            System.out.print(cs + ",");
        }
        System.out.println();
        //测试 Client 代码
        Client client = new Client();
        System.out.println("Client 函数 handle 调用节点功能");
        client.handle(root);
        //测试删除
        root.remove(new Composite("node2"));
        System.out.println("删除一个节点后: " + root.getChild());
    }
}
```

运行结果:

```
打印根节点下的所有节点
leaf1,leaf2,node1,leaf4,node3,leaf3,node2,
Client 函数 handle 调用节点功能
leaf:leaf1
leaf:leaf2
component:node1
leaf:leaf4
component:node3
leaf:leaf3
component:node2
component:root
删除一个节点后: [node1]
```

通过 Client 中 main 函数定义这个树状数据结构的代码,可以看到 root、node、leaf 的操作方法一致,组成的树就是 root,它和其他的 node 和 leaf 操作并没有什么不同。

(1) 定义一个根节点：Component root=new Composite("root");。

(2) 定义一个普通节点：Component node1=new Composite("node1");。

(3) 定义一个叶节点：Component leaf1=new Leaf("leaf1");。

(4) 将叶节点挂接到普通节点上：node1.add(leaf1);。

(5) 将普通节点挂接到另一个节点上：root.add(node1);。

所有的节点都是 Component 类型，挂接节点的方法都是使用 add 方法，想定义更复杂的树结构，只需 add 节点即可，多么的统一、和谐。

Client 中的 handle 函数参数是一个 Component 类型，它实际上就是一个树，通过组合模式形成的复杂的树状结构依然是一个 Component 类型，无论树结构多么复杂，handle 函数都不用改变。

例 5-56 树状结构遍历子节点的方法实现。

```
List<Component> result = new ArrayList<Component>();
//递归获得所有的子节点
private void find(Component c) {
    List<Component> ls = c.getChild();
    for (Component o : ls) {
        if (o instanceof Composite)
            find(o);
        result.add(o);
    }
}
```

find 方法是通过递归的方法遍历所有节点，如果子节点不是叶节点，那么继续递归遍历节点，将所发现的所有节点都放置在 result 集合中；find 方法的参数是 Component，表明 find 可以查找任意节点下的所有子节点，所以当前节点下的所有子节点代码：find(this);。

这种面向对象设计方法形成的树，虽然代码较多，但是非常清晰、直观，更简单且易于维护和扩展。

思考：

以上是通过递归方式遍历的子节点。试着实现：

(1) 使用非遍历方式获得所有子节点。

(2) 实现先序、中序及后序遍历所有子节点。

5.16 内部类和匿名类

内部类(Inner Class)：就是定义在一个类内部的类。

例 5-57 内部类的定义方法。

```
package hust.inner;

public class OuterClazz {
    public class InnerClazz{          //定义在 OuterClazz 内部的 InnerClazz 类
        public void innerFunction() {
            System.out.println("inner class");
        }
    }
```

```
    //声明在 OuterClazz 中的 InnerClazz 变量
    public InnerClazz innerClazz = new InnerClazz();

    public void outerFunction() {
        innerClazz.innerFunction();                          //使用 InnerClazz 对象中的函数
    }
}
```

上例中内部类的使用和普通类的用法并无不同，内部类的特殊性只是定义在一个类的内部，从而带来了访问外部类的一些特权，这是其他普通类所不具备的。

（1）内部类可以访问外部类中所有的元素。

```
package hust.inner;

public class OuterClazz {
    private int outer_a;
    public int outer_b;
    protected int outer_c;
    public static int OUTER_D;

    public void outerFunction() {
        System.out.println("outer function");
    }
    public class InnerClazz {
        public int outer_a;                                  //与外部类的变量同名
        public void innerFunction() {
            System.out.println(outer_a);
            System.out.println(OuterClazz.this.outer_a);     //获得外部类实例的变量
            System.out.println(outer_b);
            System.out.println(outer_c);
            System.out.println(OUTER_D);
        }
    }
}
```

上例中，可以看出内部类可以调用外部类所有的元素，当变量或者函数出现冲突时，可以显式使用 OuterClazz.this 方式获得外部对象的引用。

（2）内部对象的创建依赖外部对象，只有外部对象创建之后，才能创建内部对象，内部对象自动就会获得指向外部对象的引用。

```
//Client.java
package hust.other;

import hust.inner.OuterClazz;

public class Client {
    public static void main(String[] args) {
        OuterClazz outerClazz = new OuterClazz();
        OuterClazz.InnerClazz innerClazz = outerClazz.new InnerClazz();
    }
}
```

通过外部类对象去 new 一个内部类，使用 outerClazz.new，声明内部类也不能直接使用

内部类的类名，而需要使用 OuterClazz. InnerClazz。

含有内部类的类型被编译之后，会生成类似这样的文件名：

```
OuterClazz $ InnerClazz.class
OuterClazz.class
```

外部类会独立编译成一个文件，另一个文件 $ 前面是外部类，后面是内部类，如果内部类嵌套，会以此类推。

```
package hust.inner;

public class OtherOuterClazz {
    private class InnerClazz1 {}
    private class InnerClazz2 {
        private class InnerInnerClazz {}
    }
}
```

编译后 class 文件列表：

```
OtherOuterClazz $ InnerClazz2 $ InnerInnerClazz.class
OtherOuterClazz $ InnerClazz2.class
OtherOuterClazz $ InnerClazz1.class
OtherOuterClazz.class
```

> 内部类不允许有静态成员变量和方法(静态内部类除外)。

(3) 静态内部类称为嵌套类(static nested class)，它的创建不需要先创建外部类。

```
//OuterClazz.java
public class OuterClazz {
    private int outer_a;
    public int outer_b;
    protected int outer_c;
    public static int OUTER_D;

    public void outerFunction() {
        System.out.println("outer function");
    }

    public static class InnerClazz {                          //静态内部类
        public int outer_a;                                   //与外部类的变量同名
        public void innerFunction() {
            System.out.println(outer_a);
            //System.out.println(OuterClazz.this.outer_a);    //错误,不可访问非静态元素
            //System.out.println(outer_b);                    //错误,不可访问非静态元素
            //System.out.println(outer_c);                    //错误,不可访问非静态元素
            System.out.println(OUTER_D);
        }
    }
}
```

```
//Client.java
package hust.other;
import hust.inner.OuterClazz;

public class Client {
    public static void main(String[] args) {
        OuterClazz.InnerClazz innerClazz = new OuterClazz.InnerClazz();
    }
}
```

可以看到,创建静态内部类的方式只需要 new 即可,不需要通过外部类对象。

> 静态内部类不能访问外部类的非静态元素。

内部类和普通类用法相同,也可以继承、转型、多态等。由于内部类定义在一个类的内部,它的实现可以被完全隐藏,防止客户继承从而导致细微的破坏。

例 5-58 私有内部类的定义。

```
//OuterClazz.java
package hust.inner;

public class OuterClazz {
    public InnerClazz inner() {
        return new InnerClazz();
    }

    private class InnerClazz {                                    //隐藏内部类的实现
        public void innerFunction() {
            System.out.println("inner function");
        }
    }
}
```

将内部类定义为 private,可以防止外界客户直接继承。通过一个外部类 inner 函数返回 InnerClazz 的实例,但是实际使用过程中,会发现有些问题。

例 5-59 无法访问私有内部类。

```
//Client.java
package hust.other;
import hust.inner.OuterClazz;

public class Client {
    public static void main(String[] args) {
        OuterClazz outerClazz = new OuterClazz();
        OuterClazz.InnerClazz innerClazz = outerClazz.inner();//无法声明 innerClazz
    }
}
```

客户根本无法声明 InnerClazz 类型,因为它在定义时已经被声明为 private,外界无法使用,也就无法声明变量。

为了达到既可以隐藏实现,又可以正常使用的目的,通常使用接口。

例 5-60 隐藏类型的实现，返回接口的实例。

```
//InnerInterface.java
package hust.inner;

public interface InnerInterface {
    void innerFunction();
}

//OuterClazz.java
package hust.inner;

public class OuterClazz {
    public InnerInterface inner() {
        return new InnerClazz();
    }

    private class InnerClazz implements InnerInterface {
        public void innerFunction() {
            System.out.println("inner function");
        }
    }
    }

    //Client.java
    package hust.other;

    import hust.inner.InnerInterface;
    import hust.inner.OuterClazz;

    public class Client {
    public static void main(String[] args) {
        OuterClazz outerClazz = new OuterClazz();
        InnerInterface innerInterface = outerClazz.inner();
        innerInterface.innerFunction();
    }
}
```

由于接口是一个共有的服务协议，它被定义为 public，内部类实现后，通过外部类 inner 函数返回接口的具体实例，客户只能通过接口访问内部对象的实例，无法直接实例化，也无法继承，这样就巧妙地隐藏了实现。

> 这种隐藏实现的用法很常见，通常用于针对特定情况编写的代码，这些代码不具有通用性，开发者也不打算暴露这些实现的细节时。

一个很必要使用内部类的例子如图 5-41 所示。

在网页中绘制一个分页效果，需要一个当前页面的数据集合，用于绘制用户所要查看的数据。还需要一个数据集合的最大值，用于绘制具体的分页组件。

设计的基本类关系如图 5-42 所示。

图 5-41　网页中的分页标签

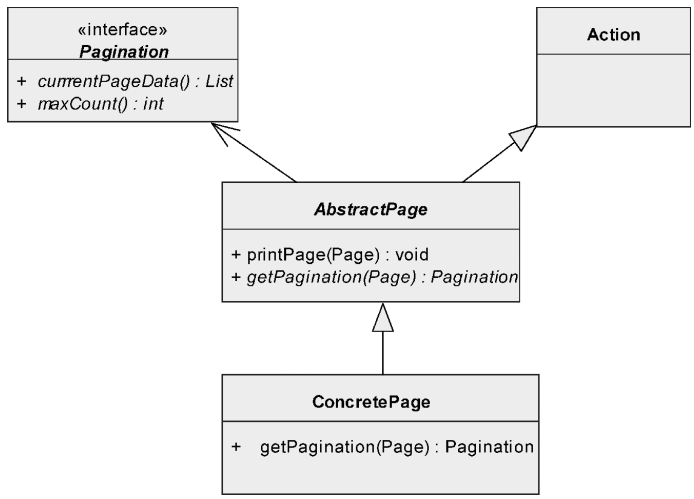

图 5-42　分页标签的类关系图

例 5-61　实现的分页标签。

```
//Pagination.java
package hust.inner;
import java.util.List;

public interface Pagination {
    List currrentPageData();
    int maxCount();
}

//AbstractPage.java
package hust.inner;
import java.util.List;

public abstract class AbstractPage extends Action {
    void printPage(Page page) {
        Pagination pagination = getPagination(page);
        List data = pagination.currrentPageData();
        int maxCount = pagination.maxCount();
        //根据获得的当前页面数据和数据库中数据的总数,
        //进行页面的分页绘制工作
    }

    protected abstract Pagination getPagination(Page page);
```

```
}

//ConcretePage.java
package hust.inner;
import java.util.List;

public class ConcretePage extends AbstractPage {
    protected Pagination getPagination(Page page) {
        return new ConcretePagePagination(page);              //返回 Pagination 对象
    }

    private class ConcretePagePagination implements Pagination {
        private Page page;

        public ConcretePagePagination(Page page) {
            this.page = page;
        }

        public List currrentPageData() {
            //在数据库中根据 page 变量传递的信息获取数据
            List data = getDataByPage(page);
            return data;
        }

        public int maxCount() {
            //根据 page 信息获得符合条件的最大数据量
            int max = getMaxByPage(page);
            return max;
        }
    }
}
```

由于基础框架的开发人员设计了接口 Pagination，返回当前页面数据和符合条件的数据最大数量值，满足了绘制页面的基本需要。同时定义了一个绘制页面的 AbstractPage，它根据客户返回的数据来进行页面的数据处理和绘制。

AbstractPage 完成了大部分通用的绘制功能，只把具体的数据提取工作留给了子类。

ConcretePage 就是一个具体的分页数据类，继承于 AbstractPage，是用户根据不同情况查询得到的特有页面，所以处理这样的分页，代码不具有通用性，所以将其中的 getPagination 函数返回的接口类型定义为内部类来处理此时的特殊情况。如果将其定义为普通类型也可以实现数据提取工作，但是毕竟普通类型定义之后，其他所有人都可以看见和使用，这样特殊用途的类型暴露在外面也没有必要。

一个更适合内部类的情况是：如果分页时，需要访问 ConcretePage 中一些内部私有数据，来进行分页数据的复杂处理，那么别无他法，只能使用内部类实现了。ConcretePagePagination 内部类具有访问 ConcretePage 中任意元素的特权，这为它深度参与 ConcretePage 逻辑、实现自己的功能提供了很大的便利。

内部类的定义位置很灵活，根据不同的目的和访问特权，可以将内部类定义在类中、函数中、代码段中。

例 5-62 内部类的灵活定义。

```
//InnerDefined.java
package hust.inner;
```

```
public class InnerDefined {
    //定义在一个类中
    private class InnerClazz1 {
    }

    public void f() {
        //定义在函数中
        class InnserClazz2 {
        }

        for (int i = 0; i < 10; i++) {
            //定义在一个作用域中
            class InnerClazz3 {
                //在内部类中在定义一个内部类
                class InnerClazz4 {
                }
            }
        }
    }
}
```

5.16.1 匿名类

没有名称的类称为匿名类。

难道还有没有类名称的类吗？构成类的一个必要条件就是应该有类名，但是有时候根据需要，定义和创建子类的实例工作同时进行，就可以省略类的名称。

例 5-63 一个普通的内部类的使用方法。

```
//AnonymousTest.java
package hust.inner;

public class AnonymousTest {
    public Service getService() {
        return new ConcreteService();
    }

    private class ConcreteService implements Service {
        public void cook() {
            System.out.println("cook");
        }
    }
}

interface Service {
    void cook();
}
```

对于返回一个接口 Service 的实现，可是通过定义一个内部类 ConcreteService，然后返回它的实例，但是如果在 getService 函数中有一个局部变量 temperature，cook 需要访问。

例 5-64 内部类不能访问局部变量的示例。

```
public class AnonymousTest {
```

```
    public Service getService() {
        int temperature = 100;
        return new ConcreteService();
    }

    private class ConcreteService implements Service {
        public void cook() {
            //内部类访问不到 temperature 变量
            System.out.println(temperature);                    //错误
        }
    }
}
```

此时 cook 是访问不到 getService 中的局部变量的。

为了访问到 temperature，可以将内部类定义在 getServcie 中。

例 5-65 将内部类定义到可以访问局部变量的位置。

```
public class AnonymousTest {
    public Service getService() {
        int temperature = 100;

        private class ConcreteService implements Service {
            public void cook() {
                System.out.println(temperature);
                System.out.println("cook");
            }
        }
        return new ConcreteService();
    }
}
```

但是这样定义依然很繁琐，ConcreteService 的存在实际上没有任何用处，它仅仅是为了定义而存在，其他任何地方都没有使用，也无法使用。如果能够将定义 ConcreteService 步骤省略，直接在返回的时候顺便实现 Service 不是更好吗！

例 5-66 使用匿名内部类简化代码实现。

```
public class AnonymousTest {
    public Service getService() {
        final int temperature = 100;
        return new Service() {
            public void cook() {
                System.out.println(temperature);
                System.out.println("cook");
            }
        };                                  //记住：这里的分号不可丢失，它是 return 语句的结束
    }
}
```

本身单独的 Service 是不允许独立 new 的，因为它是一个抽象的接口，必须通过实现其中的所有方法，它才变得有意义。而本例中 return 这个语句就是在 new Service 接口的同时给予 Service 一个具体的实现，这个实现实际上就是 Service 的一个子类，只是这个子类没有名字。

在匿名类访问局部变量 temperature 时，要求 temperature 必须是 final。但是访问成员变量时没有这个 final 的要求。

局部变量必须是 final，这保证了内部类不能更改外部的逻辑状态，防止不可控的情况出现。局部变量是函数运行过程中的状态，内部类访问的外部局部变量本来就是为了自己的业务逻辑，不能在完成自己的业务逻辑的同时更改外部的运行状态，这样可能会造成外部函数的不稳定甚至逻辑错误，所以要求内部类访问的局部变量必须是 final 的，防止被更改。

这种在实例化类型的同时给出其类型的具体实现，但没有给出实现后的具体类名称，这种特殊的类称为匿名类。

匿名类的语法看似奇怪，其实不难理解和使用。它实际上就是简化了定义一个具体类型然后再实例化的过程，将其变成了类型定义和实例化一体的过程。匿名类可以用于所有的类和接口上。

例 5-67 创建匿名内部类的同时重写函数。

```
//AnonymousTest.java
package hust.inner;

public class AnonymousTest {
    public Service getService() {
        return new Service() {
            public void cook() {
                System.out.println("anonymous test cook");
            }
        };
    }
}

class Service {
    void cook() {
        System.out.println("service cook");
    }
}
```

如果 Service 是一个类，而且已经有了一个 cook 实现，也依然可以使用匿名类实例化它的一个子类实例。getService 在返回 Service 实例的同时，可以重写 cook 方法。这样的设计，实际在 return 的同时，它完成定义 Service 的子类，并重写 cook 方法一整套流程，只是在这个过程中看不到子类的名称。

匿名类没有名称，所以它也不存在构造函数。

但是如果要实现的类型(Service)只有非默认构造函数，那匿名类只是在 new 的同时给出符合基类构造函数要求的参数即可。

例 5-68 匿名内部类的构造函数只是一个形式，无法使用。

```
package hust.inner;
```

```
public class AnonymousTest {
    public Service getService() {
        //传递 100,符合 Service 要求,但是无法更改
        return new Service(100) {
            public void cook() {
                System.out.println("anonymous test cook:" + temp);
            }
        };
    }
}

class Service {
    protected int temp = 0;
    public Service(int temp) {
        this.temp = temp;
    }
    void cook() {
        System.out.println("service cook:" + temp);
    }
}
```

temp 变量是 Service 通过构造函数初始化时赋值的,它的子类也可以通过构造函数进行修改,但是匿名类没有构造函数,也无法使用构造函数,只能自动调用父类的构造函数完成初始化工作。

内部类使 Java 可以实现类似多重继承的效果。

> 多重继承的典型特点是:
> (1) 可以向上转型为多个基类。
> (2) 在多重实现的过程中,可以重用父类中的元素,联合父类功能实现具体逻辑。

Java 可以通过接口实现多重继承的效果,但是如果基类是类的话,就只能使用内部类来实现了。使用内部类实现多重继承。

例 5-69 使用内部类实现多重继承效果。

```
//Parent.java
package hust.inner2;

public class Parent {
    protected int gender = 10;

    public void f() {
        System.out.println("parent");
    }
}

class Parent1 {
    protected int age = 20;

    public void f1() {
        System.out.println("parent1");
    }
}
```

```
class Parent2 {
    protected String name = "inner";

    public void f2() {
        System.out.println("parent2");
    }
}

class Child extends Parent {
    private class Child1 extends Parent1 {
        public void f1() {
            Child2 child2 = new Child2();
            System.out.println(child2.name);
            System.out.println(gender);
        }
    }

    private class Child2 extends Parent2 {
        public void f2() {
            Child1 child1 = new Child1();
            System.out.println(child1.age);
            System.out.println(gender);
        }
    }

    public void f() {
        Child1 child1 = new Child1();
        System.out.println(child1.age);
        Child2 child2 = new Child2();
        System.out.println(child2.name);
    }

    Parent1 getParent1() {
        return new Child1();
    }

    Parent2 getParent2() {
        return new Child2();
    }

    public static void main(String[] args) {
        Child child = new Child();
        Parent parent = child;                          //子类可以被认为是 Parent
        parent.f();
        Parent1 parent1 = child.getParent1();           //子类可以被认为是 Parent1
        parent1.f1();
        Parent2 parent2 = child.getParent2();           //子类可以被认为是 Parent2
        parent2.f2();

    }
}
```

从上述代码可以看到，Parent、Parent1、Parent2 都是 class，子类 Child 只能继承一个

Parent,其他重用 Parent1、Parent2 的操作只能通过内部类完成,Child1、Child2 是内部类,它们共同参与了 Child 的建设,从这些子类中的 f 函数可以看到,它们之间彼此都可以互相使用。这显然是多重继承带来的效果,但是在这里,使用内部类完成了同样的效果。

但是要注意的是:对于没有直接继承关系的 Child、Child1、Child2,它们之间那些受保护的 protected 元素,是无法彼此直接使用的,这与真正的多重继承相比,是有一些局限性的。

通过此种方法可以达到多重继承的效果,各个具体的函数都可以彼此调用父类的功能完成复杂的逻辑,实现的效果和真正的多重继承并没有太大的差别。

> 通常情况下,不会想到使用内部类,一般只有在以下情况下使用:
> (1) 需要有多重继承效果的。
> (2) 一个类的实现需要访问另一个类内部元素的。
> (3) 实现的类需要隐藏,不想让客户直接使用的。
> 这些情况下,才需要使用内部类。

5.16.2 回调

另一种使用匿名内部类的有趣方式就是可以构成函数的回调,函数的回调可以大大简化程序的编写。

回调机制是一种常见的设计模型,它把工作流内的某个功能按照约定的接口暴露给外部使用者,为外部使用者提供数据,或要求外部使用者提供数据。

例 5-70 使用内部类完成函数的调用。

```
package hust.callback;

public class Client {
    public void clientService(Callback callback) {
        System.out.println("begin");
        callback.f();
        System.out.println("doing");
        callback.g();
        System.out.println("end");
    }

    public static void main(String[] args) {
        Client client = new Client();

        class ConcreteCallback implements Callback {
            public void f() {
                System.out.println("callback f");
            }

            public void g() {
                System.out.println("callbakc g");
            }
        }
        client.clientService(new ConcreteCallback());
    }
}
```

```
interface Callback {
    void f();
    void g();
}
```

Client 类型中的 clientService 是一个复杂的算法，它的参数是 Callback 类型，算法中会使用到 Callback 接口中的方法参与逻辑运算，这个运算中将部分内容提交给回调函数，以便于外部使用者提供具体的逻辑。

Client 的 main 函数中，在调用 clientService 函数之前，定义一个内部类 ConcreteCallback 类型，然后传递给 clientService 函数，ConcreteCallback 中的函数 f 和 g 就称为回调函数。

f 和 g 参与了 clientService 的逻辑实现，它们是 clientService 函数的一部分功能，在函数运行过程中，clientService 将这部分功能提交给 Callback 接口让外部使用者提供一部分逻辑功能或者传递信息给外部使用者。

运行结果：

```
begin
callback f
doing
callbakc g
end
```

还可以有一种更为简洁的用法，将 main 函数更改为如例 5-70 所示的样式。

例 5-71 使用匿名内部类完成函数的调用，形成一种函数回调效果。

```
public static void main(String[] args) {
    Client client = new Client();
    client.clientService(new Callback() {
        public void f() {
            System.out.println("callback f");
        }

        public void g() {
            System.out.println("callbakc g");
        }
    });
}
```

将整个匿名类作为函数的参数，其中的接口在传递的过程中同时给出具体函数实现。这些函数在具体的逻辑算法中的适当时机会被调用，参与运算。这是一种更直接、更简洁的函数回调。

可以看出在 clientService 的逻辑运算过程中，根据算法的需要，在适当的地方调用了 Callback 接口中的 f 和 g 函数参与运算。

> 实际上这种所谓的回调也仅仅是由于使用了内部类或者匿名类而变得稍微与众不同而已，它们本质依然是普通的函数调用，Callback 也仅仅是一个函数的普通参数。由于使用了内部类或者匿名类导致这些回调函数有了更大的访问局部变量或者其他一般情况访问不到的元素的权力，这样才有了回调函数特殊的实用地位。通过匿名类实现函数回调也是比较正规的用法。

很多成熟的框架中都是用了回调，简化客户的使用，一个复杂的算法大部分被隐藏，只是暴露了需要客户填写的部分，通过接口的形式规范用户的调用。

例 5-72 Spring Framework 中的 JdbcTemplate 类中的 query 函数。

```
public void getListRowMapper() {
    ApplicationContext context =
        new FileSystemXmlApplicationContext("src/database_config.xml");
    JdbcTemplate jt =
        new JdbcTemplate((DataSource) context.getBean("oracleDataSourceTest"));
    List list = jt.query(
            "select * from region where rownum < 10",
            new RowMapper() {
                public Object mapRow(ResultSet rs, int index)
                                                    throws SQLException {
                    Map u = new HashMap();
                    u.put("region_id", rs.getString("region_id"));
                    u.put("region_name", rs.getString("region_name"));
                    return u;
                }
    });

    Iterator it = list.iterator();
    while (it.hasNext()) {
        Map map = (Map) it.next();
        System.out.println(map.toString());
    }
}
```

query 函数中的 RowMap 参数就是使用了回调，算法回调了 mapRow 函数，它通过 mapRow 函数明确 rs 数据集中数据列的对应关系及数据类型，这样 query 函数在具体的逻辑运算中可以调用 mapRow 函数得到这个客户期望的对应关系，根据这个对应关系整理数据库中的数据集合，得到了一个客户更容易使用的 list 数据集合。客户在 query 函数运算过程中参与了算法的逻辑运算。

5.17 异常处理

异常指运行期出现没有预料到的问题，导致程序出现错误。

异常处理是 Java 中唯一处理异常的机制。基本结构关系如图 5-43 所示。

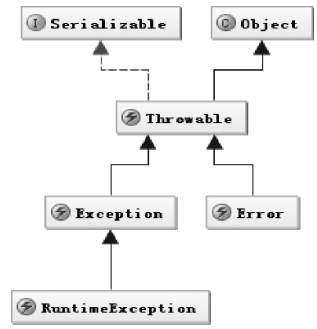

图 5-43 异常的基本结构关系

(1) Error 表示恢复不是不可能、但是很困难的情况下的一种严重问题。比如说内存溢出。不可能指望程序能处理这样的情况。

(2) Exception 表示一种设计或实现问题。也就是说，如果程序运行正常，从不会发生的情况。它分为被检异常(Checked Exception)和运行时异常(Runtime Exception)。

异常处理提供了一种程序在出现错误时能够通过某种手段恢复的机制，使程序更加健壮、稳定。Java 异常处理机制提供了统一的处理异常问题的方法。

当异常出现后：

> （1）即使修复了，所得到的值也无法正确参与后续的逻辑运算，这样的异常出现后，只能完全中断程序执行，这种异常称为运行时异常。
>
> （2）异常修复后，可以参与后续逻辑运算，修复的结果有意义，异常可以不必完全中断程序，修复的结果可以使程序继续执行，这种异常称为被检查异常。

例 5-73　可能出现异常的字符串处理程序。

```
public String firstCap(String message) {
    String[ ] words = message. split(" ");
    StringBuilder sb = new StringBuilder();
    for (String word : words) {
        sb. append(word. substring(0,1). toUpperCase());
        sb. append(word. substring(1));
        sb. append(" ");
    }
    return sb. toString();
}
```

运行结果：

```
hello world ↙
Hello World
```

这是一个能够将用户输入的英文单词首字母变为大写的函数。

用户输入的字符串：hello world，算法将句子用空格拆分为字符串数组["hello", "world"]，然后遍历数组中的单词，提取每个单词的首字母，变为大写之后，再连接在一起，返回给客户。

正常情况下，函数工作得很好，但是当用户输入：hello__world，单词之间出现两个空格时，算法拆分单词时会出现["hello","","world"]，出现了一个空字符串单词，算法在处理这个空字符串时，提取首字母 word. substring(0,1)，会出现字符串索引越界的异常：

```
Exception in thread "main" java. lang. StringIndexOutOfBoundsException: String index out of range: 1
```

如果早期预料到这样的问题，可以将代码修改一下：

```
for (String word : words) {
    if(word. equals("")) continue;
    sb. append(word. substring(0,1). toUpperCase());
    sb. append(word. substring(1));
    sb. append(" ");
}
```

但是依然会有问题，当用户输入 null 字符串，函数依然会出现错误：

```
Exception in thread "main" java. lang. NullPointerException
```

将代码修改为：

```
public String firstCap(String message) {
    if(message == null) return message;
```

```
    //其他代码
}
```

现在,算法似乎已经完善了,但是如果算法复杂,这样一一判断,代码将变得杂乱无章,掩盖了有用的代码,可读性变差,同时带来的就是可维护性变差。

上述示例代码不是异常处理,而是程序逻辑的一部分,是为了防止程序出错。真正的异常处理是在开发期间,人为的、预见性地判断异常可能出现的代码段,捕获异常并进行处理的程序。

在实际的开发过程中,程序员应该判断常见的错误,并使用正常的逻辑进行处理,不能简单地将异常情况推给用户去处理。

> 最佳实践:对于程序的安全性,不要试图检查出所有疑似问题,你一定会漏掉某些方面。应该做的是禁止所有东西,然后仅允许一些已知安全的东西通过!

5.17.1 异常的分类

```
public int div(int a,int b) {
    return a/b;
}
```

这个函数很容易想到被0除的问题,但是,这么明显的可能出现的异常Java并没有显式地提示开发人员处理,代码编译顺利通过。

```
public int div(String a,String b) {
    int aa = Integer.parseInt(a);
    int bb = Integer.parseInt(b);
    return aa/bb;
}
```

同样,Integer.parseInt(a)代码如果用户输入"2.3"同样会解析错误,系统也没有在编译期提示处理错误代码。

下面这段代码却强制开发人员进行异常处理:

```
public void write(String filename,String text) {
    Writer writer = null;
    writer = new FileWriter(filename);
    writer.write(text);
    writer.flush();
    writer.close();
}
```

看着似乎不会有什么错误,但是这样的代码在Java中不会编译通过,因为没有处理FileWriter规定的异常。将代码更改为例5-73所示。

例5-74 完成了异常处理的文件操作。

```
public void write(String filename,String text) {
    Writer writer = null;
    try {
        writer = new FileWriter(filename);
        writer.write(text);
```

```
            writer.flush();
        } catch (IOException e) {
            e.printStackTrace();
        }finally {
            if(writer!= null)
                try {
                    writer.close();
                } catch (IOException e) {
                    e.printStackTrace();
                }
        }
    }
```

要求的异常都经过了处理，编译才能通过。异常处理代码增加了代码的复杂性，相对而言，程序变得不清晰了。

从这几个例子中可以看到 Java 中处理异常的区别：

> (1) 被检查异常强制要求处理。
>
> 被检查异常要求开发者必须处理，经过处理后，程序依然可以继续运行。
>
> 被检查异常通常有警告、提示之意。通知调用者，自己提供的服务可能出现的问题和满足的条件，调用者根据这些异常提示进行相应处理，处理后，保证程序的正常运行。
>
> (2) 运行期异常不强制处理。
>
> 运行期异常是严重的异常，它可能导致系统停止运行，不强制开发者处理，因为这类异常通常处理了，可能也无法达到继续运行的要求。虽然编译器没有强制处理异常，但是如果使用者意识到了可能出现的异常问题，采取处理措施也是可以的。

运行时异常通常用于严重警告，它不在编译期提示，而是在运行期抛出，并终止程序运行，因为这类异常发生了，即表示问题非常严重，继续运行也没有意义。

Java 提供了很多异常类型，所有的异常都继承于 Throwable 类型。大多继承自 Exception 和 RuntimeException，另外还有一个表示严重错误的 Error 类型。图 5-44 显示了它们之间的关系。

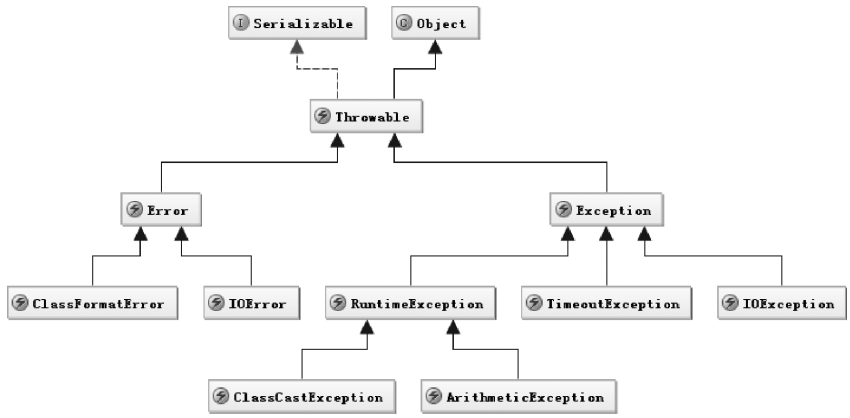

图 5-44　重要的异常类型关系图

(1) 继承自 RuntimeException 的异常类都是运行时异常，不强制客户处理。

(2) 继承自 Exception 的异常类都是被检查异常，强制客户进行异常处理。

(3) Error 表示系统严重错误。

开发时,哪些是被检异常,哪些是运行时异常,一般开发工具都可以分析出来,提示开发者,没有使用开发工具的,在编译期系统也会给出相应的未处理提示,所以这些不用全部记住。但是,记住异常这两大分类以及它们之间的区别非常重要。

可能有一个疑问:上例中在向一个文件写数据时:

```
writer = new FileWriter(filename);
```

如果文件不存在,程序会在运行期出现以下错误提示:

```
java.io.FileNotFoundException: …(系统找不到指定的路径)
```

这个异常是一个被检查异常,它在编译期就会被检测出来,catch 块捕获的异常 IOException 即提示了可能出现 I/O 错误。但实际上这样的错误一旦出现,就是很严重的问题。在文件名不存在的情况下,无法向文件写数据,如果动态给出一个默认文件名似乎意义也不大,毕竟它不是调用者的本意。那为什么还要将这么严重的异常设计为被检查异常,而不设计成运行时异常呢?

实际上这样设计,FileWriter 的设计者是希望通过这样的被检查异常提示调用者文件不存在的可能性,并尽可能地让调用者保证文件名的正确性。

如果此处设计为运行时异常,开发者接收不到任何提示信息。表示 FileWriter 的设计者无所谓文件名是否存在,只在运行期间以终止程序的方式警告调用者出现错误,然后调用者就会老老实实地去修改代码,保证程序正确运行。显然这样的方式有些不友好,将程序的错误完全推给了调用者,相比使用被检查异常就好些,它先以友好的方式提示使用者文件名如果出现错误,将会调用失败。

但使用被检查异常会使程序变得复杂,程序中会出现非常多的 try-catch-finally 语句,强制开发者进行异常处理,给开发者带来很大负担。Java 中宣称的程序更加稳定就是依赖强制客户处理异常的机制,保证可能出现的错误能得到正确处理。这个机制通过简单的 try-catch-finally 保证了程序的稳定,但是也确实增加了程序员的负担。是增加负担还是提高稳定性,仁者见仁,智者见智。

并不是完全采用运行时异常就会给程序带来好处,它倒是放松了客户处理异常的限制,但是却增加了程序出现严重异常的可能。也不能完全使用被检查异常保证程序稳定,它大大增加了开发人员的负担,导致代码混乱,不清晰,到处充斥的异常处理代码会掩盖真正的逻辑代码。

对于类库的设计者要从这些方面合理考虑使用的方便性,在保证程序稳定性的基础上,尽可能地不增加使用者的负担,是一个好的设计方向。现在的很多第三方类库、框架也朝着这个方向在做。

> 对于可恢复的条件可以使用被检查异常,对于程序错误可以使用运行时异常。

5.17.2 异常处理语法

语法

```
try{
    //可能出现异常的程序块
```

```
}catch(IllegalAccessException ex) {
    //捕获异常后的异常处理块
} catch(FileNotFoundException ex) {
    //捕获异常后的异常处理块
}finally{
    //无论异常与否,每次都保证执行的程序块
}
```

异常处理语法很简单,对要求处理或可能出现的异常程序段,使用 try{…}包围住,当异常发生时,自动使用 catch(Exception ex)语句匹配异常类型,发现匹配类型后,即会进入到异常处理程序段,程序处理完毕后,最后执行 finally 程序段,进行一些收尾工作。

catch 语句有一个异常类型的参数,像一个函数一样,只有当出现的异常与 catch 所要捕获的异常相匹配时,才会进入 catch 的异常处理程序,一旦进入 catch,其他 catch 将会被自动忽略。

被检异常和运行时异常都可以使用这个语法。

即使运行时异常没有强制要求处理,但是也可以使用这个语法进行处理。

被检异常使用这个语法处理后,如果想恢复正常逻辑,实际上依然需要更多的代码,才能使修复的结果重新参与逻辑运算。但是这样的修复代码实际上会使代码复杂性增加,很多开发人员并不愿意这样做,通常都是处理完错误之后,给出默认值或者将错误现场还原为之前的状态,然后让程序继续执行,或直接返回,用友好的方式通知调用者,执行失败,让调用者检查输入参数是否正确。这样使用被检异常虽然违背了此类异常的设计初衷,但是在保证程序稳定和代码复杂性之间也算一个折中的方案。

5.17.3 抛出异常

使用关键字 throw 同时后面跟所要抛出的异常对象就可以抛出一个异常。

```
public void div(int a,int b) {
    if(b == 0)
        throw new IllegalArgumentException("b 参数不能为 0!");
    return a/b;
}
```

此时,Java 编译器要求必须标明函数实现体中可能抛出的异常。

如果没有标明异常,客户在使用过程中出现异常就可能没有被处理,导致异常出现。

(1) 如果抛出的异常是运行时异常,函数签名不强制做特别的标识(但标识也可以)。
(2) 如果程序抛出的是被检查异常,函数签名则必须进行标识。

例 5-75 函数实现中抛出异常时,函数的定义上必须显式标明可能抛出的异常。

```
public void open(String filename) throws FileNotFoundException,IllegalAccessException {
    File file = new File(filename);
    if(!file.exists()) {
        throw new FileNotFoundException(filename + " 文件不存在");
    }
    if(!file.canRead()) {
        throw new IllegalAccessException(filename + " 文件不能访问!");
    }
}
```

出现被检查异常,需要在函数签名上使用 throws 关键字标识出函数实现体中出现的被检查异常类型,使用逗号分隔。这样的签名,调用者在调用 open 函数时,将会被要求强制处理这两个可能出现的异常。

```
public static void main(String[ ] args) {
    ExceptionTest test = new ExceptionTest();
    try {
        test.open("c:/test.txt");
        //test.open("c:\\test.txt");
    } catch (FileNotFoundException e) {
        //处理代码
    } catch (IllegalAccessException e) {
        //处理代码
    }
}
```

被检查异常强制使用 try-catch 语句将 open 函数可能出现的异常全部处理。

所有异常都是继承自 Throwable,都是可以使用 throw 关键字抛出的。

> throw 是抛出异常关键字,throws 是在函数上标识可能抛出的所有异常。

5.17.4 自定义异常

所有异常都是可抛出的(Throwable),都是继承自 Throwable,如图 5-45 所示。

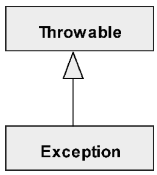

图 5-45 所有异常都继承于 Throwable

(1) 继承自 Exception,就可以自定义一个被检查异常。

```
public class FileNotReadException extends Exception {
}
```

(2) 继承自 RuntimeException 就可以自定义一个运行时异常。

```
public class FileNotReadException extends RuntimeException {
}
```

异常的类名称要有意义、直观、易读,因为异常类型本身并没有包含太多有用的信息。下面这段代码是 Java 提供的 Exception 代码。

例 5-76 Java 的 Exception 源代码。

```
//Exception.java
package java.lang;

public class Exception extends Throwable {
    public Exception() {
        super();
    }

    public Exception(String message) {
        super(message);
    }

    public Exception(String message,Throwable cause) {
```

```
        super(message,cause);
    }

    public Exception(Throwable cause) {
        super(cause);
    }
}
```

从 Java 提供的 Exception 基类来看，只有 message 这一个能够说明异常详细信息的变量。更多地去扩充异常的其他信息也没有太多意义，当然，Java 并没有阻止用户扩展 Exception。

重新定义 open 函数，使用 FileNotReadException。

```
public void open(String filename) throws FileNotReadException {
    File file = new File(filename);
    if(!file.canRead()) {
        throw new FileNotReadException (filename + " 文件不能读!");
    }
}
```

再次使用 open 函数：

```
public static void main(String[ ] args) {
    ExceptionTest test = new ExceptionTest();
    try {
        test.open("c:/test.txt");
        //test.open("c:\\test.txt");
    } catch (FileNotReadException e) {
        //处理代码
    }
}
```

自定义异常和普通系统自带的标准异常没有任何区别，使用方法也相同。

> 在开发过程中尽量使用系统的标准异常，这些标准异常命名非常标准、易懂，所有程序员都很熟悉。

5.17.5 异常的归并

在开发过程中，根据系统功能的需要，经常需要将底层的多个异常进行统一归并，再次向上抛出，屏蔽底层的低级异常。

一个访问数据库的 JDBC 代码，需要处理很多异常。

例 5-77 访问数据库时，将众多低级异常归并为一个高级异常。

```
//DatabaseUtility.java
package hust.exception;
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.SQLException;
import java.sql.Statement;

public class DatabaseUtility {
```

```
    public void insert(String sql) throws HighException {
        Connection conn = null;
        Statement st = null;
        try {
            Class.forName("com.mysql.jdbc.Driver");
            conn = DriverManager.getConnection(
        "jdbc:mysql://localhost:3306/test","root","123");
            st = conn.createStatement();
            st.execute(sql);
        } catch (ClassNotFoundException e) {
            //异常处理
            HighException high = new HighException("驱动程序异常,请检查!");
            high.initCause(e);
            throw high;
        } catch (SQLException e) {
            //异常处理
            HighException high =
            new HighException("insert 语句执行失败,错误代码: " + e.getErrorCode());
            high.initCause(e);
            throw high;
        } finally {
            if (st != null) {
                try {
                    st.close();
                } catch (SQLException e) {
                    //异常处理
                }
            }
            if (conn != null) {
                try {
                    conn.close();
                } catch (SQLException e) {
                    //异常处理
                }
            }
        }
    }

    public static void main(String[] args) {
        DatabaseUtility low = new DatabaseUtility();
        try {
            low.insert("insert into student values(1,2,3)");
        } catch (HighException ex) {
            ex.printStackTrace();
        }
    }
}

//HightException.java
package hust.exception;

public class HighException extends RuntimeException {
    public HighException(String message) {
```

```
        super(message);
    }
}
```

如果 insert 函数的设计者不处理程序抛出的两个异常 ClassNotFoundException 和 SQLException，那么 insert 的签名上将会多出两个客户需要处理的异常：

```
public void insert(String sql) throws ClassNotFoundException,SQLException
```

但是对于客户来说，处理这两个异常有些莫名其妙，因为执行一条语句，竟然要求处理两个异常，而且异常过于底层，很不友好。

当 insert 异常发生时，将所有的异常归并为一个 HighException，并给出更直观的提示，让用户不直接接触底层的 SQL 异常，对于客户来说很友好。

HightException 根据需要被定义为 RuntimeException，不强制客户处理。

程序发生异常时，运行结果：

```
hust.exception.HighException: 驱动程序异常,请检查!
    at hust.exception.DatabaseUtility.insert(DatabaseUtility.java:19)
    at hust.exception.DatabaseUtility.main(DatabaseUtility.java:48)
    at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
    at sun.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccessorImpl.java:39)
    at sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorImpl.java:25)
    at java.lang.reflect.Method.invoke(Method.java:597)
    at com.intellij.rt.execution.application.AppMain.main(AppMain.java:120)
Caused by: java.lang.ClassNotFoundException: com.mysql.jdbc.Driver
    at java.net.URLClassLoader$1.run(URLClassLoader.java:200)
    at java.security.AccessController.doPrivileged(Native Method)
    at java.net.URLClassLoader.findClass(URLClassLoader.java:188)
    at java.lang.ClassLoader.loadClass(ClassLoader.java:303)
    ... 6 more
```

可以看出异常的抛出层次，它首先在

```
Caused by: java.lang.ClassNotFoundException: com.mysql.jdbc.Driver
```

处抛出，之后被转换为高级的异常 HighException 异常，再次抛出

```
hust.exception.HighException: 驱动程序异常,请检查!
```

在归并底层异常时，最好将异常的层次结构保留，以便于调试，查找错误。保留异常的层次结构需要使用 initCause(e)函数进行封装，异常便可以再次封装重新抛出。

```
catch (ClassNotFoundException e) {
        //异常处理
        HighException high = new HighException("驱动程序异常,请检查!");
        high.initCause(e);
        throw high;
}
```

将 ClassNotFoundException 异常的堆栈信息填充进 HighException 中，被重新抛出，这样对外屏蔽了 ClassNotFoundException。

下面的代码并没有归并异常，只是将异常简单地抛出：

```
catch (ClassNotFoundException e) {
        //异常处理
        throw e;
}
```

这样的设计没有增加新的异常层次结构。

运行结果：

```
java.lang.ClassNotFoundException: com.mysql.jdbc.Driver
    at java.net.URLClassLoader$1.run(URLClassLoader.java:200)
    at java.security.AccessController.doPrivileged(Native Method)
    at java.net.URLClassLoader.findClass(URLClassLoader.java:188)
    at java.lang.ClassLoader.loadClass(ClassLoader.java:303)…
```

可以看到就是一层异常结构，没有增加新的异常。这样的处理只是简单地将异常抛给了使用者。

通过将多种低级异常进行归并，可以大大减少客户处理异常的代码量。在设计通用框架时，通常要使用多种第三方的类库，这些类库通常都有自己的异常结构，代码的重用，使框架也集成了多种异常结构，这些都属于框架的设计和实现的细节，不应该暴露给具体的使用客户，所以此时需要将各种底层的异常通过新的异常结构设计，重新抛出，以一种和框架设计相关的、友好的方法展示给使用者，而不是简单地直接抛出，这是一个好的设计。

流行的 Spring Framework 整合众多的异常结构，使之都屏蔽在了 Spring 自己的架构异常之下，Spring 以一种新的、统一的、更高级的异常展示给客户。这样，使用者只看到了和 Spring 架构相关的异常，而不用知道底层具体的第三方的异常结构，这会使代码在处理异常上更加统一。

5.17.6 捕获异常的顺序

例 5-78 异常的捕获顺序示例。

```
//ExceptionCatch.java
package hust.exception;

public class ExceptionCatch {
    public static void main(String[] args) {
        try {
            throw new D();
        }catch (C e) {
            System.out.println("C catch:" + e);
        }catch (B e) {
            System.out.println("B catch:" + e);
        }catch (A e) {
            System.out.println("A catch:" + e);
        }
    }
}

class A extends Exception {}
class B extends A {}
```

```
class C extends B {}
class D extends C {}
```

运行结果：

```
C catch:hust.exception.D.
```

当 try 中程序抛出异常 D 时，catch 进行捕获，按照声明的顺序依次进行类型匹配，如果不匹配，那么向上转型再次匹配，如果依然不匹配将会执行下一个 catch 语句，再次匹配。只要匹配上一个 catch 语句，以下的 catch 语句将会被忽略，如果最终没有找到匹配的异常，那么这个异常将会直接抛出，程序会发生运行时异常。

此时异常 D 首先匹配 C，D 是 C 的子类，当然可以被认为是 C 类型，所以异常匹配，程序会直接进入 catch(C e) 异常处理程序段。其他的两个异常处理程序段将会被忽略。

这样的匹配规则对多个 catch 语句中捕获的异常类型顺序就有相应的限制了。

例 5-79 错误的异常捕获用法。

```
public static void main(String[] args) {
    try {
        throw new D();
    }catch (A e) {
        System.out.println("A catch:" + e);
    }catch (B e) {
        System.out.println("B catch:" + e);
    }catch (C e) {
        System.out.println("C catch:" + e);
    }
}
```

同样的代码，但是将捕获顺序更改为 A、B、C，程序将会编译错误，因为当抛出的异常 D 去匹配 catch 时，永远都会匹配到第一个 catch 语句，后面两个捕获 B、C 的 catch 语句永远都不会被执行。因为 B 和 C 都是 A 的子类，无论抛出 B、C、D 异常都会被当做 A 异常捕获，后面捕获的 B 和 C 异常永远被 A 阻挡，无法执行到。

> 综上，在捕获的异常有继承关系时，要先捕获子类异常，然后才是父类异常，这样能够保证所有 catch 语句都有效。

catch 语句只能捕获到 try 中发生的异常，没有发生的异常进行 catch 捕获无效，编译也不会通过。

例 5-80 错误地捕获从不可能发生的异常。

```
public static void main(String[] args) {
    try {
        throw new F();
    }catch (C e) {
        System.out.println("C catch:" + e);
    }catch (B e) {
        System.out.println("B catch:" + e);
    }catch (F e) {
        System.out.println("A catch:" + e);
```

```
    }
}

//F.java
public F extends Exception {}
```

main 函数中，try 程序段只抛出了 F 异常，catch C，B 语句都是无效的，编译错误。

5.17.7 子类抛出异常

> 子类不可以抛出比父类更多的异常，但子类可以抛出比父类少的异常。

例 5-81 子类抛出了比父类更多的异常。

```
interface Service {
    void cook() throws HotException;
}

class ChildService implements Service {
    private int temperature = 10;

    public void cook() throws HotException,ColdException {
        if (temperature > 100)
            throw new HotException("温度太高!");
        if (temperature < 0)
            throw new ColdException("温度太低!");

        System.out.println("正常");
    }
}

class HotException extends Exception {
    public HotException(String message) {
        super(message);
    }
}

class ColdException extends Exception {
    public ColdException(String message) {
        super(message);
    }
}
```

ChildService 中的 cook 函数是错误的，因为它抛出了比父类 Service 更多的异常，这会带来一个问题：针对父类编程的代码只处理了 Service 抛出的异常，它们认为不会再有其他的异常出现了，但是如果子类还能抛出更多的异常，直接导致客户代码少处理了这些新的异常，运行期就会发生错误，这种情况 Java 是不允许出现的，在编译器就会检查出错误，并提示：hust.exception.ChildService 中的 cook() 无法实现 hust.exception.Service 中的 cook()；被覆盖的方法不抛出 hust.exception.ColdException。

客户代码：

```
public static void main(String[] args) {
    Service service = new ChildService();
    try {
        service.cook();
    } catch (HotException e) {
        System.out.println("cook 出现了异常");
    }
}
```

针对 Service 的编程中，在调用 cook 函数时，客户代码只知道 cook 抛出了 HotException，它不会知道有 ColdException 抛出，如果抛出了 ColdException，那这段代码就将出现错误。

> 如果子类实现的 cook 函数不抛出任何异常，这样不会影响客户代码对异常的处理，所以子类可以抛出比父类少的异常。

下面子类实现的 cook 代码是正确的。

例 5-82 子类可以抛出比父类少的异常。

```
class ChildService implements Service {
    private int temperature = 10;

    public void cook() {                                    //未抛出任何异常
        System.out.println("正常");
    }
}
```

建议通常情况下不要放大，也不要缩小父类的异常。以下代码是合适的：

```
class ChildService implements Service {
    private int temperature = 10;

    public void cook() throws HotException {
        if (temperature > 100)
            throw new HotException("温度太高!");
        if (temperature < 0)
            System.out.println("温度太低,请做其他处理!");

        System.out.println("正常");
    }
}
```

所以：

> (1) 子类不可以抛出比父类更多的异常。
> (2) 抛出的异常可以是父类抛出异常的子类。

5.17.8 finally 关键字

finally 程序段是每次都会保证执行的，无论异常发生与否。通常异常发生时，try 程序段中的后续语句将不会被执行，系统直接跳转到 catch 段执行异常处理程序，但是如果在 try 中打开一些资源，如何进行清理和关闭呢？

DatabaseUtil 类中 insert 函数。

例 5-83 有潜在问题的数据库访问代码。

```
public void insert(String sql) throws Exception {
    Connection conn = null;
    Statement st = null;
    try {
        Class.forName("com.mysql.jdbc.Driver");
        conn =
DriverManager.getConnection("jdbc:mysql://localhost:3306/test","root","123");
        st = conn.createStatement();
        st.execute(sql);
        st.close();                                          //出现异常,代码不会被执行
        conn.close();                                        //出现异常,代码不会被执行
    } catch(ClassNotFoundException e) {
        //异常处理
    } catch (SQLException e) {
        //异常处理
    }
}
```

如果异常发生在 st.execute(sql)处,系统将立即进行 catch (SQLException e)异常处理程序,那么后续的两个关闭语句 st.close();conn.close();永远都不会被执行,statement 和 connection 所占用的资源将无法释放。

finally 程序段保证了无论异常发生与否,都肯定会执行。finally 给程序执行必要的清理工作提供了技术保障。

finally 必须与 try 一起使用。可以不必有 catch。

```
try {

}finally {

}
```

所以 DatabaseUtil 类中的 insert 函数关闭 statement 和 connection 的工作应该被写到 finally 程序段中。

例 5-84 正确地访问数据库和关闭数据库的实现代码。

```
public void insert(String sql) throws Exception {
    Connection conn = null;
    Statement st = null;
    try {
        Class.forName("com.mysql.jdbc.Driver");
        conn = DriverManager.getConnection(
                        "jdbc:mysql://localhost:3306/test","root","123");
        st = conn.createStatement();
        st.execute(sql);
    } catch (ClassNotFoundException e) {
        //异常处理
        HighException high = new HighException("数据库驱动异常!");
```

```
            high.initCause(e);
            throw high;
        } catch (SQLException e) {
            //异常处理
            HighException high =
                new HighException("insert 语句执行失败,错误代码: " + e.getErrorCode());
            high.initCause(e);
            throw high;
        } finally {
            if (st != null) {
                try {
                    st.close();
                } catch (SQLException e) {
                    //异常处理
                }
            }
            if (conn != null) {
                try {
                    conn.close();
                } catch (SQLException e) {
                    //异常处理
                }
            }
        }
    }
```

将 close 数据库连接的操作移到 finally 程序段中后,保证了无论异常发生与否,都肯定会释放数据库资源。

例 5-85　finally 和 return 的执行顺序示例。

```
//FinallyTest.java
package hust.exception;

public class FinallyTest {
    private int count = 0;
    public int print(String info, int returnValue) {
        System.out.println(info + returnValue);
        return returnValue;
    }
    public int show() {
        try{
            if(1 == 1)
                throw new Exception();
            return print("try return, value = ", ++count);
        }catch (Exception e) {
            return print("catch return, value = ", ++count);
        }finally {
            return print("finally return, value = ", ++count);
        }
    }

    public static void main(String[] args) {
        FinallyTest test = new FinallyTest();
```

```
        System.out.println(test.show());
    }
}
```

运行结果：

```
catch return,value = 1
finally return,value = 2
2
```

从这段测试程序可以看出，finally 程序段肯定最后会被执行，而且即使 catch 捕获异常返回之后，也会执行 finally 程序段。

try/catch 中的 return 语句调用的函数先于 finally 中调用的函数执行，也就是说，return 语句先执行，finally 语句后执行。所以，返回的结果是 2。return 并不是让函数马上返回，而是 return 语句执行后，将把返回结果放置进函数栈中，此时函数并不是马上返回，它要执行 finally 语句后才真正开始返回。

将 show 函数简化：

```
public int show() {
    try {
        int i = 1;
        if(i == 1)
            throw new Exception();
        return 0;
    }catch (Exception e) {
        return - 1;
    }finally {
        return - 2;
    }
}
```

简化的 show 函数编译后的字节码：

```
0 iconst_1
 1 istore_1
 2 iload_1
 3 iconst_1
 4 if_icmpne 15 ( + 11)
 7 new # 10 < java/lang/Exception >
10 dup
11 invokespecial # 11 < java/lang/Exception.< init >>
14 athrow
15 iconst_0
16 istore_2
17 bipush - 2
19 ireturn
20 astore_1
21 iconst_m1
22 istore_2
23 bipush - 2
25 ireturn
26 astore_3
```

```
27 bipush -2
29 ireturn
```

bipush 指令是把 byte 类型的值压入栈顶，ireturn 指令在返回之前会从操作数栈中获取返回值。

运行结果：

```
-2
```

思考： final、finalize、finally 单词看着很相似，它们有什么区别吗？

5.18 泛型

如果想开发一个非常通用的程序，那么，不针对具体类型编写的代码是最具一般性的。

例 5-86 针对基本 Object 编程的极为通用的函数。

```
//Client.java
package hust.generic;

public class Client {
    public void show(Object obj) {
        System.out.println(obj);
    }

    public static void main(String[] args) {
        Client client = new Client();
        client.show(1);
        client.show("hello");
        client.show(new Object());
        client.show(new Client());
    }
}
```

Client 中的 show 函数参数为 Object，它可以支持 Java 中的所有类型，所以 show 函数极为通用。

上述 show 函数只是简单地打印，如果 show 函数的实现体中想调用参数的具体行为，那么将变得很困难，因为 Object 本身没有太多实际意义的服务，都是一些通用的抽象服务。

例如，如果 show 函数想调用参数对象中的 print 函数，Object 并不具备。

上述代码说明：

> 通用代码也是相对而言的，不能一般化到无意义的参数对象。

如果想让 show 函数可以调用参数对象中的 print 函数，同时又不太多限制 show 函数的通用性，可以使用接口完成这个意图。

例 5-87 使用接口完成一定程度的通用函数设计。

```
//Client.java
package hust.generic;
```

```
public class Client {
    public void show(GenericInterface obj) {
        obj.print();
    }

    public static void main(String[] args) {
        Client client = new Client();
        client.show(new GenericInterface() {
            public void print() {
                System.out.println("interface print");
            }
        });
    }
}

interface GenericInterface {
    void print();
}
```

这样 show 函数支持所有基于 GenericInterface 接口的类型，可以在 show 的实现体中调用参数对象的 print 函数，去实现 show 的一些复杂功能。这样做确实限制了 show 函数的通用性，但却增加了 show 函数的可用性。show 函数只支持 GenricInterface 类型，但是基于接口的编程，放松了对类型的限制，客户可以轻松地将现有类型转换到符合 show 要求的参数。

例 5-88 通过实现接口可以重用已有代码。

```
//Client.java
package hust.generic;

public class Client {
    public void show(GenericInterface obj) {
        obj.print();
    }

    public static void main(String[] args) {
        Client client = new Client();

        class ConvertService extends OtherService implements GenericInterface {
            public void print() {
                this.write();
            }
        }
        client.show(new ConvertService());
    }
}

interface GenericInterface {
    void print();
}

class OtherService {
    void write() {
        System.out.println("other service");
```

```
    }
}
```

show 函数参数使用 GenericInterface 接口，一点都没有阻碍重用其他代码。接口没有限制一个已经使用继承关系的类型去再次实现它，这就增加了 show 函数的通用性。

OtherService 类和 GenericInterface 接口没有任何关系，也无法直接应用到 show 函数中，但是通过继承 OtherService 和实现 GenericInterface 接口，就可以重用 OtherService 中的 write 函数，同时又能够适应 show 函数的需要，此处如果 GenericInterface 不是接口，将会限制 show 函数的通用性。

例 5-89 使用匿名内部类重用 OtherService。

```
public class Client {
    public void show(GenericInterface obj) {
        obj.print();
    }

    public static void main(String[] args) {
        Client client = new Client();

        client.show(new GenericInterface() {
            public void print() {
                new OtherService().write();
            }
        });
    }
}
```

使用匿名内部类也可以达到重用 OtherService 类中 write 函数的目的。

思考：如果 GenericInterface 是一个抽象类或者普通类型，那么还可以使用什么方法重用 OtherService 呢？

例 5-90 接口函数返回值为了通用而设计成 Object 类型。

```
//Client.java
package hust.generic;

public class Client {
    public void show(GenericInterface genericInterface) {
        System.out.println(genericInterface.get().getClass().getName());
    }

    public static void main(String[] args) {
        Client client = new Client();
        client.show(new ConcreteObjectOne(10));
        client.show(new ConcreteObjectTwo("hello"));
    }
}

interface GenericInterface {
    Object get();
}
```

```
class ConcreteObjectOne implements GenericInterface {
    private int age;

    public ConcreteObjectOne(int age) {
        this.age = age;
    }

    public Object get() {
        return age;
    }
}

class ConcreteObjectTwo implements GenericInterface {
    private String name;

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

    public Object get() {
        return name;
    }
}
```

如果一个函数返回值针对 Object 编写，例如上述接口函数，那么调用 get 函数返回的 Object 可以支持所有类型，很通用，但是所有返回值都被向上转型为 Object，以至于调用者 show 函数并不知道具体的类型是什么，无法直接使用，限制了 show 函数的逻辑实现。

是否有一种既能够支持所有类型，又能够明确具体类型的技术呢？

Java 中的泛型可以部分解决这个问题，但是依然复杂和难用，除非到了非常明确要使用泛型技术时，否则不要轻易使用泛型。

5.18.1 泛型定义

1. 泛型类

泛型类

```
public class Generic<T>{
    public void get(T t) {
        System.out.println(t);
    }
}
```

例 5-91 泛型类的定义与使用。

```
//Generic.java
package hust.generic;

public class Generic<T>{
    public void get(T t) {
        System.out.println(t);
    }
```

```
    public static void main(String[] args) {
        Generic<String> generic = new Generic<String>();
        generic.get("hello");

        Generic<Light> genericOne = new Generic<Light>();
        genericOne.get(new Light());
    }
}

class Light{
}
```

声明泛型：使用<>括号，里面写上一个泛型的名称 T，它仅仅是一个名称，它不代表任何类型，在整个类的作用范围内都可以使用 T 这种泛型类型。

使用泛型：声明时要明确 T 所代表的类型，否则编译不通过。上例声明时，一个支持了 String，一个支持了 Light 类型，get 函数的参数 T 也立刻变成了支持上述类型。因为 Generic<T>都是在声明的时候将 T 替换为具体的类型。

> 泛型就是一种不针对任何具体类型的类型，它是泛泛的类型，极度不具体的类型，所以也就没有具体的可以使用的服务。

可以看出 get 函数里面根本无法使用 T 参数中的服务，只是简单地打印。即使在运行过程中可以支持任何类型，但是在设计期却无法预知具体类型，所以使用泛型时，看到的程序通常都是针对泛型 T 的定义、转换、动态创建等通用代码，也正因为泛型 T 没有具体服务，这样的代码才非常通用，但为了达到良好的实用性，在设计上也相对复杂。

例 5-92 泛型类型只能通过强制转型才能使用。

```
//Generic.java
package hust.generic;

public class Generic<T> {
    public void get(T t) {
        if(t instanceof Light) {
            Light light = (Light) t;
            light.on();
        }
    }

    public static void main(String[] args) {
        Generic<String> generic = new Generic<String>();
        generic.get("hello");

        Generic<Light> genericOne = new Generic<Light>();
        genericOne.get(new Light());
    }
}

class Light {
    public void on() {
        System.out.println("light on");
    }
}
```

通过使用 instanceof 动态判断 T 的类型，强制向下转换，是可以使用具体的服务的。

如果类中打算支持多个泛型，可以这样声明：

```
public class Generic < A,B >{ }
```

2. 泛型接口

上述代码是在类上声明泛型，还可以在接口上声明泛型：

```
泛型接口
    public interface Generic <A,B >{
        public A get();
    }
```

例 5-93 使用泛型接口示例。

```
package hust.generic;

public class Client {
    public void show(GenericInterface genericInterface) {
        System.out.println(genericInterface.get().getClass().getName());
    }

    public static void main(String[] args) {
        Client client = new Client();
        client.show(new ConcreteObjectOne < String >("hello"));
    }
}

interface GenericInterface < T > {
    T get();
}

class ConcreteObjectOne < H > implements GenericInterface < H > {
    private H t;

    public ConcreteObjectOne(H t) {
        this.t = t;
    }

    public H get() {
        return t;
    }
}

client.show(new ConcreteObjectOne < String >("hello"));
```

show 函数在调用过程中，接收的 ConcreteObjectOne 在实例化时明确了泛型类型为 String，所以 ConcreteObjectOne 的构造函数立刻变为了以 String 为参数的构造函数。

GenericInterface＜H＞和 GenericInterface＜T＞的含义是一样的。此时的 ConcreteObjectOne 依然是一个泛型类型不确定的类，它需要在实例化时明确执行泛型的具体类型，否则无法

使用。

示例中，可以看到定义时使用 T，但在实现时却使用了 H，这是因为它们都仅仅是一个泛型标志，可以不同，但是却代表相同的含义，但是建议使用一致的名称。

```
interface GenericClazz<A,B>{
    void print(A a);
    void print(B b);
}
```

GenericClazz 中的两个 print 函数是重复的。

正是由于泛型符号的特殊性，不代表具体的类型信息，所以类中的 print 函数编译器会认为是同一个函数，并不是重载。

5.18.2 泛型的兼容

例 5-94 泛型兼容示例。

```
package hust.generic;

public class Client {

    public void show(GenericInterface genericInterface) {
        System.out.println(genericInterface.get().getClass().getName());
    }
    public static void main(String[] args) {
        Client client = new Client();
        client.show(new ConcreteObjectOne());
    }
}

interface GenericInterface<T>{
    T get();
}

class ConcreteObjectOne implements GenericInterface<String>{
    public String get() {
        return "hello";
    }
}
```

可以看到 Client 代码和普通的代码没有任何区别，但是实际上它们却隐含着使用泛型，这样的设计带来了一个好处：就是可以在不更改客户代码的情况下，让整个系统支持泛型，兼容性非常好。

ConcreteObjectOne 在实现 GenericInterface 泛型接口时，明确指定了泛型类型为 String，这样实际上 ConcreteObjectOne 已经变成了一个具体的类型，变成了 get 返回值为 String 的具体类型，在使用中和普通的类型已经没有任何分别。

5.18.3 泛型函数

```
泛型函数
    public class Generic {
```

```
        public<B>void get(B b) {
    }
}
```

注意：<B>定义时可以使用其他名称，例如

```
public class Generic {
    public <File> void get(File file) {
    }
}
```

这个 File 泛型和 Java 类库中的 File 类不是一个概念，此时的泛型 File 就是一个名称，和 B 没有任何区别，所以它也不可能具有 Java 类库中 File 类中的众多功能。

泛型函数的使用和普通的函数没有任何区别。

```
generic.get("hello");
generic.get(1);
generic.get(new Light());
```

泛型函数在使用时，感觉它是一个被无限次重载的函数，支持所有的类型。

> 从几个简单的例子可以看出，泛型似乎意义不大，主要是因为它在设计期依然没有明确类型，无法使用对象其中的具体功能，也就无法完成更复杂的逻辑功能。通常使用泛型的地方都可以使用接口来等价实现，因为接口也是一种极度抽象的、不具体的、没有任何实现的，同时也不限制类的继承关系的弱侵入性类型。

但是如果不使用泛型中的具体服务，而是针对泛型编写的通用集合类，这时，泛型的实际意义比较大。因为它使集合变得非常通用，可以支持任意类型的元素；集合通常只是持有元素，维护集合中元素的位置，执行快速查找，增加删除等操作，而不直接调用元素的具体服务。

5.18.4 泛型的边界

由于泛型没有任何服务，如果能够基于一个接口或者类来约束它的类型，那样不就可以具有一些需要的服务了吗？确实可以。

例 5-95 泛型的边界限定示例。

```
//GenericTest.java
package hust.generic;

public class GenericTest<T extends Service>{
    public static void main(String[] args) {
        GenericTest<Service> genericTest = new GenericTest<Service>();
        genericTest.callService(new ConcreteService());
    }

    public void callService(T t) {
        t.print();
    }
}
```

```
class ConcreteService implements Service {
    public void print() {
        System.out.println("print service");
    }
}

interface Service {
    void print();
}
```

通过 extends 关键字达到了类似继承的效果，这种语法约束了泛型的服务范围，只能是 Service 类型，它使泛型的通用性有了一定的限制，但是却增加了可用性。

实际上这样的代码反而使事情复杂化了，直接将 callService 函数的参数变为接口 Service，使用传统的编程方法更简单。

例 5-96 使用接口编程可以替换泛型边界限制。

```
//GenericTest.java
package hust.generic;

public class GenericTest {
    public static void main(String[] args) {
        GenericTest genericTest = new GenericTest();
        genericTest.callService(new ConcreteService());
    }

    public void callService(Service t) {
        t.print();
    }
}

class ConcreteService implements Service {
    public void print() {
        System.out.println("print service");
    }
}

interface Service {
    void print();
}
```

这里再一次否定了泛型技术。

Java 泛型边界也可以是多个，例如：

```
<T extends Service & ServiceInterfaceOne & ServiceInterfaceTwo>
```

边界只能有一个类，而且只能放在 extends 后第一个位置，后面使用 & 连接符号，其他的全部是接口。

例 5-97 多个泛型边界限制。

```
//GenericTest.java
package hust.generic;
```

```
public class GenericTest < T
                    extends Service &
                            ServiceInterfaceOne &
                            ServiceInterfaceTwo > {
    public static void main(String[ ] args) {
        GenericTest < ConcreteService > genericTest =
                new GenericTest < ConcreteService >();
        genericTest.callService(new ConcreteService());
    }

    public void callService(T t) {
        t.write();
        t.print();
        t.show();
    }
}

class ConcreteService
        extends Service
        implements ServiceInterfaceOne,ServiceInterfaceTwo {
    public void write() {
        System.out.println("print service");
    }

    public void print() {
        System.out.println("service interface one print");
    }

    public void show() {
        System.out.println("service interface two show");
    }
}

class Service {
    void write() {
        System.out.println("service write");
    }
}

interface ServiceInterfaceOne {
    void print();
}

interface ServiceInterfaceTwo {
    void show();
}
```

通过类似继承和实现的方式限制了泛型的边界，就可以按照泛型的边界类型调用相应的函数实现具体的逻辑代码，这里的设计依然可以使用传统编程方法替换，而且也会更直观。

如果 ConcreteService 定义成如下样式，将不适用于＜T extends Service & ServiceInterfaceOne & ServiceInterfaceTwo＞的边界约定：

(1) class ConcreteService ***implements*** ServiceInterfaceOne,ServiceInterfaceTwo

(2) class ConcreteService ***extends*** Service ***implements*** ServiceInterfaceOne

(3) class ConcreteService ***extends*** Service

总之缺少了泛型规定的任何一个元素,都不符合 callService 函数 T 参数的要求。

但是元素除了这些,再多些,是符合约定的,例如:

```
class ConcreteService extends Service
              implements ServiceInterfaceOne,
                         ServiceInterfaceTwo,
                         Cloneable
```

ConcreteService 实现时,又多实现了一个 Cloneable 接口,这对于 callService 来说,无所谓,此时 ConcreteService 依然符合 callService 函数参数的要求。

显然 extends 规定了泛型的一个边界,实际上它是一个上边界。它规定泛型参数只有它规定的类型及其子类才符合要求。

5.18.5 简单的集合类

Java 本身提供了强大的集合类型,在没有介绍集合之前,可以使用数组自己创建一个基本的集合类。

例 5-98 一个简单的集合类型。

```
//CollectionTest.java
package hust.generic;

public class CollectionTest {

    public static void main(String[] args) {
        SimpleCollection collection = new SimpleCollection(5);
        //测试自动扩展性
        for (int i = 0; i < 65; i++) {
            collection.add("hello" + i);
        }
        System.out.println(collection.size());
        System.out.println(collection.delete(10));
        System.out.println(collection.delete(30));
        System.out.println(collection.delete(40));
        //测试边界
        collection.add(61,"bad");
        collection.add(62,"good");
        //测试边界扩展性
        collection.add(64,"good1");
        collection.add(64,"good2");
        //测试边界
        collection.add("good3");
        //测试边界
        System.out.println(collection.delete(collection.size() - 1));
        //测试打印
        System.out.println(collection);
        //测试自动收缩性
        for (int i = 0; i < 30; i++) {
            collection.delete(0);
```

```
            }
            System.out.println(collection);
            //测试 get
            System.out.println(collection.get(23));
            System.out.println(collection.size());
        }
    }

    interface Collection {
        void add(Object item);
        void add(int index,Object item);
        Object delete(int index);
        Object get(int index);
        int size();
    }

    class SimpleCollection implements Collection {
        //自动扩展的大小
        private static final int AUTO_INC_SIZE = 20;
        //集合容器
        private Object[] array;
        //集合中有效对象的数量
        private int size;

        //初始化指定大小容器
        public SimpleCollection(int size) {
            array = new Object[size];
            this.size = 0;
        }

        //默认大小容器
        public SimpleCollection() {
            array = new Object[AUTO_INC_SIZE];
            this.size = 0;
        }

        //在末尾添加对象
        public void add(Object item) {
            add(size,item);
        }

        //在指定位置插入对象
        public void add(int index,Object item) {
            if (index > size)
                throw new ArrayIndexOutOfBoundsException
                        ("超出索引:" + index + ",最大:" + (size - 1));
            //有效对象数量超出容器大小,自动扩充容器
            if (array.length == size) {
                Object[] result = new Object[array.length + AUTO_INC_SIZE];
                for (int i = 0; i <= array.length; i++) {
                    if (i < index) {
                        result[i] = array[i];
                    } else if (i > index) {
```

```
                result[i] = array[i - 1];
            }
        }
        result[index] = item;
        array = result;
    } else {
        //插入位置后的对象整体后移
        for (int i = size; i > index; i--) {
            array[i] = array[i - 1];
        }
        array[index] = item;
    }
    size++;
}

//删除指定位置对象
public Object delete(int index) {
    if(index > size - 1)
        throw new ArrayIndexOutOfBoundsException
              ("超出索引:" + index + ",最大:" + (size - 1));
    Object delObject = null;
    //无效对象过多,容器自动收缩
    if(array.length - size > AUTO_INC_SIZE * 2) {
        Object[] result = new Object[size + AUTO_INC_SIZE];
        for (int i = 0; i < size; i++) {
            if(i < index) {
                result[i] = array[i];
            }
            if(i > index)
                result[i - 1] = array[i];
        }
        size--;
        delObject = array[index];
        array = result;
    }
    //删除位置后对象前移
    if (index < size) {
        for (int i = index; i < size; i++) {
            if ((i + 1) < array.length)
                array[i] = array[i + 1];
        }
        //移动后清除末尾无用对象
        array[size - 1] = null;
        size--;
        return array[index];
    }
    return delObject;
}

public Object get(int index) {
    if (index < size) {
        return array[index];
    }
```

```
            throw new ArrayIndexOutOfBoundsException
                    ("超出索引:" + index + ",最大:" + (size - 1));
        }

        public int size() {
            return size;
        }

        public String toString() {
            StringBuilder sb = new StringBuilder();
            sb.append("[");
            for (int i = 0; i < size; i++) {
                sb.append(array[i]);
                if(i < size - 1)
                    sb.append(",");
            }
            sb.append("]");
            return sb.toString();
        }
    }
```

这个集合使用了最基本的数组进行实现,甚至连操作数组的 Java 工具类也没有使用。从这段代码中可以学习到数组的使用、元素的复制和移位操作。一个简单的集合类的基本功能有添加、删除、获得对象等,该集合本身可以根据需要自动扩展和收缩大小。

Collection 是一个集合接口,它定义了一般集合中需要的方法,SimpleCollection 是这个接口的一个实现。

从代码可以看出,一个简单的集合类型编写的代码也比较复杂,如果考虑线程安全、效率等需要,代码将更复杂。Java 本身提供了非常稳定、效率很高,而且线程安全的集合类型,通常,开发人员不用自己去编写一个基本的集合类,只需重用强大的 Java 集合类型即可。

上例代码提供了一个可以容纳所有类型的集合,因为容器在设计时,其中的集合元素被设计为 Object,表示它支持所有的 Java 类型。这样一个集合非常通用,从中也可以看到对容器中的对象没有任何直接操作,都只是在容器中移动、删除对象等。

这样的代码存在一个不方便之处:

(1) 所有对象被放置到容器中时,对象都会自动向上转型为 Object。

(2) 提取的对象类型将不可获知。

(3) 而且由于容器支持所有对象,容器中的对象很可能不是一种类型。

例 5-99 集合的使用。

```
//Client.java
package hust.generic;

public class Client {
    public static void main(String[] args) {
        Collection collection = new SimpleCollection();
        //容器可以容纳任何类型
        collection.add("hello");
        collection.add(1);
        collection.add(new Client());
```

```
        System.out.println(collection.get(0) instanceof String);
        System.out.println(collection.get(1) instanceof Integer);
        System.out.println(collection.get(2) instanceof Client);
        //可能出现错误
        int sum = 10 + Integer.parseInt(collection.get(1).toString());
        System.out.println(sum);
    }
}
```

运行结果：

```
true
true
true
11
```

在计算 sum 值时，从集合中获得索引 1 位置的对象，可能出现错误，该位置不一定是整型。

为了限制容器中对象的类型，可以重用 SimpleCollection，编写一个针对特定类型元素的容器。

例 5-100 只能容纳 Client 类型的集合容器。

```
package hust.generic;

public class Client {
    private int tag;

    public Client(int tag) {
        this.tag = tag;
    }

    public String toString() {
        return "client tag = " + tag;
    }

    public static void main(String[] args) {
        ClientCollection clientCollection = new ClientCollection();
        //clientCollection.add(1);                                    //出错
        clientCollection.add(new Client(1));
        clientCollection.add(new Client(2));
        clientCollection.add(new Client(3));
        //获得的对象类型直接可以获知，不用强制转换
        Client client = clientCollection.get(0);
        System.out.println(client.tag);

    }
}

class ClientCollection {
    private Collection simpleCollection = new SimpleCollection();

    public void add(Client item) {
        simpleCollection.add(item);
```

```
    }

    public void add(int index,Client item) {
        simpleCollection.add(index,item);
    }

    public Client delete(int index) {
        return (Client) simpleCollection.delete(index);
    }

    public Client get(int index) {
        return (Client) simpleCollection.get(index);
    }
}
```

实现一个针对 Client 对象的容器，由于重用了 SimpleCollection 集合功能，这个 ClientCollection 集合实现起来相对简单得多。

实际上这个 ClientCollection 集合虽然具有集合的所有功能，但它并不是一个 Collection，因为它没有从 Collection 继承，当然也不可能声明为：

```
Collection collection = new ClientCollection();                    //错误
```

例如存在一个面向 Collection 类型的函数：

```
public void show(Collection collection) {
    for (int i = 0; i < collection.size(); i++) {
        System.out.println(collection.get(i));
    }
}
```

ClientCollection 将不能用于这个函数！这潜在地限制了 ClientCollection 的重用。

为了实现只支持特定元素类型的容器，系统只能设计出众多的这种类似 ClientCollection 的类型，代码重复，相似。

思考：将 ClientCollection 也定义成为一个 Collection，同时也重用了 SimpleCollection 集合的功能，该如何做呢？

5.18.6 理想的对象容器

理想容器有几个要求：

(1) 能够支持所有类型。

(2) 能够限制插入指定类型的对象。

(3) 提取的对象不用转型，即可明确对象类型。

(4) 容量自动扩展，能够随时获得对象的数量。

实现一个泛型集合就可以达到这样的目的，将上述代码更改为支持泛型的集合。

例 5-101 使用泛型改进自定义的集合容器。

```
//CollectionTest.java
package hust.generic;

public class CollectionTest {
```

```
    public static void main(String[] args) {
        Collection<Client> clientCollection = new SimpleCollection<Client>();
        clientCollection.add(new Client(1));
        clientCollection.add(new Client(2));
        clientCollection.add(new Client(3));
        Client client = clientCollection.get(1);
    }
}

interface Collection<T> {
    void add(T item);
    void add(int index,T item);
    T delete(int index);
    T get(int index);
    int size();
}

class SimpleCollection<T> implements Collection<T> {
    //自动扩展的大小
    private static final int AUTO_INC_SIZE = 20;
    //集合容器
    private T[] array;
    //集合中有效对象的数量
    private int size;

    //初始化指定大小容器
    public SimpleCollection(int size) {
        array = (T[]) new Object[size];
        this.size = 0;
    }

    //默认大小容器
    public SimpleCollection() {
        array = (T[]) new Object[AUTO_INC_SIZE];
        this.size = 0;
    }

    //在末尾添加对象
    public void add(T item) {
        add(size,item);
    }

    //在指定位置插入对象
    public void add(int index,T item) {
        if (index > size)
            throw new ArrayIndexOutOfBoundsException
                    ("超出索引:" + index + ",最大:" + (size - 1));
        //有效对象数量超出容器大小,自动扩充容器
        if (array.length == size) {
            T[] result = (T[]) new Object[array.length + AUTO_INC_SIZE];
            for (int i = 0; i <= array.length; i++) {
                if (i < index) {
                    result[i] = array[i];
```

```
                } else if (i > index) {
                    result[i] = array[i - 1];
                }
            }
            result[index] = item;
            array = result;
        } else {
            //插入位置后的对象整体后移
            for (int i = size; i > index; i--) {
                array[i] = array[i - 1];
            }
            array[index] = item;
        }
        size++;

    }

    //删除指定位置对象
    public T delete(int index) {
        if(index > size - 1)
            throw new ArrayIndexOutOfBoundsException
                    ("超出索引:" + index + ",最大:" + (size - 1));

        T delObject = null;
        //无效对象过多,容器自动收缩
        if(array.length - size > AUTO_INC_SIZE * 2) {
            T[] result = (T[]) new Object[size + AUTO_INC_SIZE];
            for (int i = 0; i < size; i++) {
                if(i < index) {
                    result[i] = array[i];
                }
                if(i > index)
                    result[i - 1] = array[i];
            }
            size--;
            delObject = array[index];
            array = result;
        }
        //删除位置后对象前移
        if (index < size) {
            for (int i = index; i < size; i++) {
                if ((i + 1) < array.length)
                    array[i] = array[i + 1];
            }
            //移动后清除末尾无用对象
            array[size - 1] = null;
            size--;
            return array[index];
        }
        return delObject;
    }

    public T get(int index) {
```

```
        if (index < size) {
            return array[index];
        }
        throw new ArrayIndexOutOfBoundsException
              ("超出索引:" + index + ",最大:" + (size - 1));
    }

    public int size() {
        return size;
    }

    public String toString() {
        StringBuilder sb = new StringBuilder();
        sb.append("[");
        for (int i = 0; i < size; i++) {
            sb.append(array[i]);
            if(i < size - 1)
                sb.append(",");
        }
        sb.append("]");
        return sb.toString();
    }
}
```

这个泛型集合显示了它的优越性,它支持所有的类型,而且能够保证容器中对象类型的一致性,这样在提取时也就可以确切地获知对象类型了。

```
Collection<Client> clientCollection = new SimpleCollection<Client>();
```

在声明时,指定容器中的对象类型是 Client 类型,那么它的所有方法 add、delete、get 都只能支持 Client 类型。这里可以看出使用泛型技术的强大威力,它创造了一个万能的集合,太令人兴奋了。

例 5-102 泛型集合的灵活应用。

```
//SimpleCollection.java
package hust.generic;

public class SimpleCollectonTest {
    public static void main(String[] args) {
        SimpleCollection<A> simpleCollectionA = new SimpleCollection<A>();
        simpleCollectionA.add(new B());
        A a = simpleCollectionA.get(0);
    }
}

class A{}
class B extends A{}
```

这样做没有任何问题,声明一个支持 A 类型的集合,放入的却是 B 类型。因为 B 从 A 派生而来,本身是可以被当做 A 来看待的。

但是下面的代码确实有问题:

```
SimpleCollection<A> simpleCollectionA = new SimpleCollection<B>();  //错误
```

声明一个支持 A 类型的集合，实例化的却是使用的 B 类型。正常理解，应该没有任何问题，B 本身就可以完全当做 A 对待，为什么不能实例化成 B 集合呢？

实际上这里涉及的不是容器中对象是否兼容，而是两个集合是否兼容，SimpleCollection＜A＞和 SimpleCollection＜B＞实际上就是两个不同类型的集合，两个不同的集合对象不可以互相赋值，在两个不同类型的集合上讨论集合中的对象是否兼容没有意义。这就像“猫＝狗”，这样赋值是不正确的，讨论其中的毛发是否兼容也没有意义。

那上述代码如何更改，使它们兼容呢？只能使用通配符。

5.18.7 通配符

```
SimpleCollection<? extends A> simpleCollectionA = new SimpleCollection<B>();
Collection<? extends A> collection = new SimpleCollection<B>();
```

这样的通配符声明是正确的。它表明了声明的集合支持 A 边界泛型类型的元素，B 显然受 A 的约束。

使用通配符使基于泛型的函数变得更加通用。

例 5-103 使用通配符使泛型函数更通用。

```
//SimpleCollectionTest.java
package hust.generic;

public class SimpleCollectonTest {
    public void operation(Collection<? extends A> collection) {
        for (int i = 0; i < collection.size(); i++) {
            A a = collection.get(i);
            a.show();
        }
    }

        public static void main(String[] args) {
        SimpleCollectonTest test = new SimpleCollectonTest();
        Collection<A> collectionA = new SimpleCollection<A>();
        collectionA.add(new A());
        collectionA.add(new B());
        test.operation(collectionA);
        Collection<B> collectionB = new SimpleCollection<B>();
        //collectionB.add(new A()); //错误
        collectionB.add(new B());
        test.operation(collectionB);
    }
}

class A {
    public void show() {
        System.out.println("A");
    }
}
class B extends A {
    public void show() {
```

```
            System.out.println("B");
        }
    }

    class C extends A {
        public void show() {
            System.out.println("C");
        }
    }

    class D extends B {
        public void show() {
            System.out.println("C");
        }
    }
```

operation 函数支持所有元素派生于 A 的集合，既支持 Collection＜A＞也支持 Collection＜B＞。所以在遍历的时候，get 出来的元素都肯定是 A 元素或者它的子类，不用强制转型即可使用。

Collection＜A＞中可以插入 A 以及它的子类作为元素。

Collection＜B＞中插入它的父类 A 作为元素是错误的。

例 5-104 危险的声明方式。

```
public static void main(String[] args) {
    SimpleCollectonTest test = new SimpleCollectonTest();
    Collection collectionB = new SimpleCollection<B>();
    System.out.println(collectionB);
    collectionB.add(1); //可以插入集合
    collectionB.add(new B());
    collectionB.add(new A());
    test.operation(collectionB);
}
```

上述代码可以编译通过，但是存在着潜在的问题：

```
Collection collectionB = new SimpleCollection<B>();
```

如此声明，会导致 collectionB 无法保证容器中的元素都是 B 类型。这样声明编译器只能推断集合中所要求的元素是 Object 类型而非 B 类型，会导致用户可以填入所有类型的元素。用户在调用 operation 函数时，内部就会出现错误，元素 1 不是 B 类型，不存在 show 函数，这样的错误只有在运行时才会发现。代码出现异常：

```
Exception in thread "main" java.lang.ClassCastException: java.lang.Integer cannot be cast to
hust.generic.A
```

> 使用泛型时，尽可能不要让编译器去推断类型，而应显式地指明运行期类型，这样会减少很多潜在的问题。

1. extends 上界

上界用 extends 关键字声明，表示参数化的类型可能是所指定的类型，或者是此类型的子类。

```
Collection<? extends A> simpleCollectionB = new SimpleCollection<B>();
Collection<? extends A> simpleCollectionC = new SimpleCollection<C>();
Collection<? extends A> simpleCollectionC = new SimpleCollection<D>();
```

这些 Collection<? extends A>对应的实例都是正确的，它们都是 A 的子类型。

这样声明后 Collection 中的 add 函数参数变为“? extends A”。

虽然这样声明是对的，但是 simpleCollectionB.add(new B())；却编译错误。查看例 5-105 这样一个等价函数。

例 5-105 基于泛型上界的函数。

```
public void operation(Collection<? extends A> collection) {
    for (int i = 0; i < collection.size(); i++) {
        A a = collection.get(i);
        a.show();
    }
}
```

operation 的参数就是一个 Collection<? extends A>参数类型。operation 循环集合中的元素没有任何问题，但是要是向其中添加元素，将出现问题。

```
public void operation (Collection<? extends A> collection) {
    collection.add(new A());                                    //错误
    collection.add(new B());                                    //错误
}
```

添加 A 和 B 的实例均编译错误。实际上无论添加什么元素都是错误的。

operation 参数的定义表明它既支持 Collection<A>也支持 Collection<B>类型的集合。那么：

```
Collection<A> collectionA = new SimpleCollection<A>();
collectionA.add(new A());                                       //正确
collectionA.add(new B());                                       //正确
Collection<B> collectionB = new SimpleCollection<B>();
collectionB.add(new A());                                       //错误
collectionB.add(new B());                                       //正确
```

它们正确的交集是 add(new B())。那么将 operation 函数改为：

```
public void operation (Collection<? extends A> collection) {
    collection.add(new B());                                    //难道正确吗?
}
```

显然道理不通。原因是：

(1) 由于 Collection<? extends A>中的 add 函数无法获知运行期到底传递过来的是 A 还是 A 的子类集合，所以无法确定 add 的具体参数，也无法确定集合中的对象类型。

(2) 如果运行期 Collection<? extends A>传递过来的是 Collection<A>，万事大吉；但如果传递过来的是 Collection<B>，添加 A 类型显然是不正确的，因为 A 不能被当做 B 看待；此时的 collection.add(new B())；倒是正确的。

(3) 但是别忘了，也可能传递过来的是 Collection<C>，C 也是 A 的子类，这时 collection.add(new B())；显然就是错误的了。

综合分析之后发现 Collection<? extends A>定义的集合，无论 add 的元素是 A 还是 A 的子类型都是不正确的。这种声明通常用于遍历确定类型的集合，而不是操作集合。

但这种方法声明的集合遍历时获得的元素类型确实可以明确获知的 A 类型，不用强制转换。

将 operation 函数改变一下：

```
public < T extends A > void operation (Collection < T > collection, T item) {
    collection.add(item);
}
```

这样就可以向集合中添加元素了，而且元素类型肯定是确定的。

> extends 不是继承，是边界限定，是一个上边界限定。

2. super 下界

下界用 super 进行声明，表示参数化的类型可能是所指定的类型，或者是此类型的父类型，直至 Object，这也称为“逆变”。

```
Collection<? super D> collectionDA = new SimpleCollection<A>();
Collection<? super D> collectionDB = new SimpleCollection<B>();
Collection<? super D> collectionDD = new SimpleCollection<D>();
```

这些声明方式是正确的，它和 extends 上界正好相反。实例化的集合泛型类型是前面声明类型 D 的父类型，这样保证了添加进集合的元素肯定是 D 及其子类型。

例 5-106 基于泛型下界的函数。

```
package hust.generic;

public class SimpleCollectonTest {
    public void operation(Collection<? super D> collection) {
        for (int i = 0; i < collection.size(); i++) {
            D a = (D) collection.get(i);                                    //0
            a.show();
        }
    }

    public static void main(String[] args) {
        SimpleCollectonTest test = new SimpleCollectonTest();
        Collection<? super A> collectionAA = new SimpleCollection<A>();     //1
        collectionAA.add(new A());                                          //2
        collectionAA.add(new B());                                          //3
        collectionAA.add(new D());
        collectionAA.add(new E());                                          //4
        Collection<? super B> collectionBB = new SimpleCollection<B>();     //5
        //collectionBB.add(new A()); //error                                //6
        collectionBB.add(new B());                                          //7
        collectionBB.add(new D());
        collectionBB.add(new E());                                          //8
        Collection<? super D> collectionDA = new SimpleCollection<A>();     //9
        //collectionDA.add(new A()); //error                                //10
        //collectionDA.add(new B()); //error
        collectionDA.add(new D());                                          //11
```

```
        collectionDA.add(new E());                                          //12
        Collection<? super D> collectionDB = new SimpleCollection<B>();     //13
        //collectionDB.add(new A()); //error                                //14
        //collectionDB.add(new B()); //error
        collectionDB.add(new D());                                          //15
        collectionDB.add(new E());                                          //16
        Collection<? super D> collectionDD = new SimpleCollection<D>();     //17
        //collectionDD.add(new A()); //error                                //18
        //collectionDD.add(new B()); //error
        collectionDD.add(new D());                                          //19
        collectionDD.add(new E());                                          //20

        //Collection<? super D> collectionDE =
                                    new SimpleCollection<E>(); //error     //21

        test.operation(collectionAA);                                       //22
        test.operation(collectionBB);                                       //23
        test.operation(collectionDA);                                       //24
        test.operation(collectionDB);                                       //25
        test.operation(collectionDD);                                       //26
    }
}

//A,B,C,D类定义同上,略
//在此基础上再添加一个E类型
class E extends D {
    public void show() {
        System.out.println("E");
    }
}
```

从声明的方式来看 Collection<? super D>只接收了元素 D 及其 D 以上类型的集合,不能接收 E 类型元素的集合。而 Collection<? extends A>正好相反,它只接收 A 及 A 以下类型作为元素的集合类型。

从这些集合添加元素的类型来看,它只能添加具体集合规定的类型及其子类型。例如:

Collection<? super D>collectionDD=new SimpleCollection<D>();它只接收 D 及以下的元素类型。Collection<? super A>collectionAA=new SimpleCollection<A>();只接收 A 及 A 以下的元素类型。

> 接收什么类型和声明直接相关,和后面变量对应的具体集合类型无关。

> Collection<? super D>限制了对应实例化的集合元素类型的下限。确定了集合的下限后,添加的元素也就确定了,它是下限 D 及其以下类型的元素。
>
> 下限保证了集合中的元素肯定符合实例化集合元素的规定。

但是有个问题,查看程序注释 0:由于 D 是下限,运行期无法获知父类是什么,所以在遍历集合中元素时,也就无法直接获得类型,它们可能是 D,也可能是 D 的父类,也可能是 D 的子类。如果是 D 的子类,直接转换成 D 没有问题,但是如果是 D 的父类(注释 22、23),那到底应该转换成哪个父类呢?无法获知。所以注释 0 处的强制转换是危险的。当传递 Collection<? super A>集合时,集合中的 A 类型是无法转换为 D 类型的,A 是 D 的父类,会

抛出异常：

```
Exception in thread "main" java.lang.ClassCastException: hust.generic.A cannot be cast to hust.
generic.D
```

所以这里要想正确使用必须使用 instanceof 运行期动态判断类型进行安全转换。

将注释 0 处代码改为 A a=(A) collection.get(i);,它将适合所有的类型，因为 A 是这些类型的父类。但是这样使用要明确知道 A 作为父类所提供的服务有限，如果能够接受这个现实，这样使用没有问题。

一个典型的例子：如果定义一个针对“汽车”的集合，而且“汽车”是下限：Collection<? super 汽车>，这样一个集合允许添加很多“汽车”及其子类型(轿车、车、货车)。它保证了无论定义什么具体的汽车集合，都能保证它是一个汽车集合，例如：

```
Collection<? super 汽车>collection = new Collection<交通工具>();
```

声明的集合依然是一个汽车集合，它保证了其中的元素是“汽车”及其子类型，它们都是“交通工具”。感觉定义“交通工具”有点大材小用：实例化了一个更通用的集合(交通工具集合)，但是确实只能添加较小范围的元素(汽车作为元素)，比汽车更一般的交通工具都不能添加。但是这种限制确实保证了声明的 Collection<? super 汽车>类型变量可以安全地添加汽车类型的元素，而上界 Collection<? extends 汽车>无法保证安全地添加元素。

本节更多是介绍 Java 中的泛型技术和概念，除实现通用泛型集合之外，一般情况下没有明显需要去使用泛型技术。泛型的使用并不是那么显而易见，只有在非常明确且没有替换方案时，使用泛型技术才是明智的。

5.19 工程实践经验总结

- 继承是两个实体间的一种关系，其中一个实体是基于另一个实体而定义的。
- 继承使父类的代码得以在子类中重用，Java 只允许实体有一个父类。
- 父类是子类的泛化类型，父类是子类的更抽象类型，父类比子类更一般化。
- 一般化父类时不要将子类的特殊功能赋予父类，否则基于父类的所有子类都将拥有这些可能无用的功能。
- 继承是为了重用。通常要先考虑通过组合方式重用代码，只有在非常必要时，而且类型之间有明确父子关系时才使用继承。
- Java 继承使用 extends 关键字定义，并且只能有一个父类。
- 子类不扩充父类的接口是一个好的习惯，它是一种 is-a 关系。不扩充带来的好处就是子类可以完全替换父类。
- 子类扩充父类后，是一种 is-like-a 关系，子类替换父类将导致子类扩充的特殊功能消失。有时候子类是需要扩充父类功能的。
- 在定义变量和函数参数时尽可能地针对父类编程，这会使代码更加通用。除非要使用子类的特有功能时，才定义基于具体类型的变量和函数参数。
- 所有子类都可以安全地向上转型。也就是说，子类可以被完全当做父类看待。
- 父类不可以安全地向下转型。因为父类可能对应多个子类型，无法确认父类到底对应的是哪个类型。

- 先通过 instanceof 可以判断对象到底是哪一个具体的子类型，然后将父类强制转换为子类，这样做是安全的。
- 子类实例化后，父类和子类是联合存在于内存当中的。即使是被重写的父类函数依然没有消失，也是可以使用的(只能在子类中使用)。外界是看不到父类被重写的原始函数的，只能看到重写过的函数。
- 子类在实例化时，先通过父类的构造函数实例化父类，而后才是实例化子类。子类要通过一种确定的方式调用父类构造函数，否则在无法实例化父类时，也就无法成功实例化子类自己了。
- super 是子类对象指向父类的引用。通过它可以调用父类的构造函数、普通函数和被覆盖的成员变量。
- super 在显式调用父类构造函数时，也只能在子类的构造函数中的第一行调用，不可以在普通的函数中调用。
- final 具有不变、不可修改之意。它修饰的类、成员变量、局部变量、函数、函数参数等都有不可修改之意。
- final 修饰的成员变量必须要进行初始化(空白 final 除外)，不能依赖系统提供的默认值。
- 重写是子类重定义父类相同签名的函数。
- equals 用于判断两个对象是否相等，必须重写相等逻辑。==符号不能判断两个对象是否相等，只能判断主类型数据是否相等。
- toString 用于以字符串形式表达一个具体的对象，它通常是具体对象的一个简单描述。重写它可以给使用者一个良好的使用体验。
- 多态是在运行时刻接口匹配的对象能互相替换的能力。它的关键是：运行时、接口匹配、替换。它是一种后期绑定机制。
- 实现多态的步骤：继承，重写，针对父类编程，运行时，接口替换。
- 多态要针对行为进行设计，而不要针对对象的状态。重写状态时，一定要知道影响的范围。
- 子类在重写父类函数时，可以返回一个比父类更具体的类型。
- 子类在重写父类函数时，函数的可视性可以比父类更高，但是不能更低。
- 即使是一个完全抽象的类型，它依然是一个类，子类继承它之后，不可以再有其他父类。
- 接口是一个全新的类型，外观看似一个完全的抽象类，但是子类却可以实现多个接口。
- 即使一个类中没有任何抽象元素，该类也可以被定义为抽象。但是如果类中有抽象元素，那么该类一定要被定义为抽象类。
- 接口可以多重继承接口。类可以多重实现接口。
- 接口中的方法默认都是公有的，接口中的变量默认都是常量，而且必须被初始化。
- 适配器模式是将原本不兼容的接口可以工作在一起。
- 策略设计模式是通过算法的独立变化，可以在运行期间被客户随意替换。
- 组合设计模式是将对象组合成树状结构以表示“部分-整体”的层次结构，它使整体与部分的操作方法一致。
- 内部类是定义在一个类内部的类型。它可以被所有的权限修饰符修饰，也可以是静态的。

- 内部类自动持有一个外部类的引用，它可以访问外部类的元素。
- 内部类可以被定义在类中、函数中、嵌套在内部类中。
- 匿名类是没有名称的类。有时实现的类型非常特殊，没有重用性，就可以在定义的同时进行实现，这样的设计达到了只能在一处使用的目的。
- 通过内部类可以达到多重继承的效果。
- 匿名类访问局部变量或函数参数时，需要变量是 final 的，防止匿名类更改。
- 匿名内部类不能有静态元素。
- 回调是匿名类的典型应用。
- 异常分为被检异常和运行时异常。被检异常强制要求处理，运行时异常不强制要求处理。
- 对于可恢复的条件可以使用被检异常，对于程序错误可以使用运行时异常。
- 所有异常都继承于 Throwable，RuntimeException 继承于 Exception。所有异常都是可抛出的。
- 抛出异常使用 throw 关键字。表示函数可能抛出的异常使用 throws 关键字。
- 子类不可以抛出比父类更多的异常，但是可以抛出比父类少的异常。
- finally 程序段是无论如何都肯定会执行的程序段。finally 即使在函数 return 之后，依然会执行。
- 泛型定义中的＜T＞只是一个符号，它可以是任何 Java 允许的字符或单词。
- 泛型类：class Generic＜T＞，class Generic＜A，B＞；泛型接口：interface GenericInterface＜T＞，泛型函数：＜B＞void get(B b)。
- ＜T extends Service＞，这里的 extends 是泛型参数的上边界。它规定的类型及其子类才符合泛型参数的要求。
- ＜? super Service＞，这里的 super 是泛型参数的下边界。只有规定类型及其他的父类符合泛型参数要求。
- 使用泛型时，尽可能不要让编译器去推断类型，而应显式地指明运行期类型，这样会减少很多潜在的问题。
- 泛型非常适合在集合上应用。

5.20 求职实战

(1) final 修饰变量时，是引用不能变，还是引用的对象不能变？

(2) ＝＝和 equals 有什么区别？

(3) overload 和 override 的区别是什么？overload 的方法是否可以改变返回值的类型？

(4) 构造函数是否可以被 override？

(5) 接口是否可以继承接口？抽象类是否可以实现接口？抽象类是否可继承具体类？抽象类中是否可以有静态的 main 方法？

(6) 如何实现多态机制？

(7) abstract class 和 interface 有什么区别？

(8) 抽象方法可以是 static 的吗？

(9) 什么是内部类？

(10) 内部类可以引用它的外部类的成员吗？有什么限制吗？

(11) static nested class 和 innner class 有什么不同？

(12) anonymous inner class 是否可以 extends 其他类，是否可以 implements 接口？

(13) try 里面有一个 return 语句，那么紧跟在这个 try 后的 finally 里面的代码会不会执行，什么时候执行，在 return 前还是后？

(14) final，finally 和 finalize 的区别是什么？

(15) 运行时异常与一般的异常有何不同？

(16) error 和 exception 有什么区别？

(17) Java 中异常处理机制的简单原理和应用是什么？

(18) 举一些常见的运行时异常类型。

(19) Java 中如何进行异常处理？关键字 throws、throw、try、catch、finally 分别代表什么意义？在 try 块中可以抛出异常吗？

(20)

```
abstract class Name {
    private String name;
    public abstract boolean isStupidName(String name) {}
}
```

有什么错误？

(21)

```
public class Something {
    void doSomething () {
        private String s = "";
        int l = s.length();
    }
}
```

有什么错误？

(22)

```
abstract class Something {
    private abstract String doSomething ();
}
```

有什么错误？

(23)

```
public class Something {
    public int addOne(final int x) {
            return ++x;
    }
}
```

有什么错误？

(24)

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

```
    Other o = new Other();
        new Something().addOne(o);
    }
    public void addOne(final Other o) {
        o.i++;
    }
}
class Other {
    public int i;
}
```

有什么错误?

(25)

```
class Something {
    int i;
    public void doSomething() {
        System.out.println("i = " + i);
    }
}
```

有什么错误?

(26)

```
class Something {
    final int i;
    public void doSomething() {
        System.out.println("i = " + i);
    }
}
```

有什么错误?

(27)

```
public class Something {
    public static void main(String[] args) {
        Something s = new Something();
        System.out.println("s.doSomething() returns " + doSomething());
    }
    public String doSomething() {
        return "Do something ...";
    }
}
```

有什么错误?

(28)

```
class Something {
    private static void main(String[] something_to_do) {
        System.out.println("Do something ...");
    }
}
```

有什么错误?

(29)

```
interface A{
    int x = 0;
}
class B{
    int x = 1;
}
class C extends B implements A {
    public void pX() {
      System.out.println(x);
    }
    public static void main(String[] args) {
        new C().pX();
    }
}
```

有什么错误?

(30)

```
interface Playable {
    void play();
}
interface Bounceable {
    void play();
}
interface Rollable extends Playable,Bounceable {
    Ball ball = new Ball("PingPang");
}
class Ball implements Rollable {
    private String name;
    public String getName() {
        return name;
    }
    public Ball(String name) {
        this.name = name;
    }
    public void play() {
        ball = new Ball("Football");
        System.out.println(ball.getName());
    }
}
```

有什么错误?

(31) 编程题:用 Java 实现二叉树。

(32) 编程题:从类似如下的文本文件中读取所有的姓名,并打印出重复的姓名和重复的次数,并按重复的次数排序:

1,张三,28

2,李四,35

3,张三,28

4,王五,35

5,张三,28

6,李四,35

7,赵六,28

8,田七,35

(33) 编程题：写出一个单态模式的代码。

(34) 编程题：一个整数,大于零,不用循环和本地变量,按照 n、2n、4n、8n 的顺序递增,当值大于 5000 时,把值按照执行的顺序打印出来,例如：n=1237。

(35) 编程题：第一个人 10 岁,第二个比第一个人大 2 岁,以此类推,第 8 个人多大。

(36) 编程题：排序有几种方法？用 Java 实现一个快速排序。

(37) 编程题：将数组元素顺序翻转。

(38) 编程题：将金额转换为中国传统形式的样式：例如￥1011 转换为壹仟零壹拾壹元整。