软·件·设·计·技·术

Java 设计模式深入研究

刘德山 金百东 ◎ 编著

人民邮电出版社
POSTS & TELECOM PRESS

Java 设计模式深入研究

刘德山 金百东 ◎ 编著

人 民 邮 电 出 版 社
北 京

图书在版编目（CIP）数据

Java设计模式深入研究 / 刘德山，金百东编著. --
北京 : 人民邮电出版社，2014.7
ISBN 978-7-115-35186-9

Ⅰ. ①J… Ⅱ. ①刘… ②金… Ⅲ. ①JAVA语言－程序
设计 Ⅳ. ①TP312

中国版本图书馆CIP数据核字(2014)第087800号

内 容 提 要

设计模式是一套被重复使用的代码设计经验的总结。本书面向有一定 Java 语言基础和一定编程经验的读者，旨在培养读者良好的设计模式思维方式，加强对面向对象思想的理解。

全书共分 12 章，首先强调了接口和抽象类在设计模式中的重要性，介绍了反射技术在设计模式中的应用。然后，从常用的 23 个设计模式中精选 10 个进行了详细的讲解，包括 2 个创建型模式、4 个行为型模式、4 个结构型模式。本书理论讲解透彻，应用示例深入。设计模式的讲解均从生活中的一类常见事物的分析引出待讨论的主题，然后深入分析设计模式，最后进行应用探究。应用探究部分所有示例都源自应用项目，内容涉及 Java、JSP、JavaScript、Ajax 等实用技术，知识覆盖面广。

本书可供高等院校计算机相关专业本科生和研究生设计模式、软件体系结构等课程使用，对高级程序员、软件工程师、系统架构师等专业研究人员也具有一定的参考价值。

◆ 编　　著　刘德山　金百东
责任编辑　邹文波
责任印制　彭志环　焦志炜

◆ 人民邮电出版社出版发行　　北京市丰台区成寿寺路 11 号
邮编　100164　　电子邮件　315@ptpress.com.cn
网址　http://www.ptpress.com.cn
北京中新伟业印刷有限公司印刷

◆ 开本：787×1092　1/16
印张：13.75　　2014 年 7 月第 1 版
字数：363 千字　　2014 年 7 月北京第 1 次印刷

定价：45.00 元

读者服务热线：(010)81055256　印装质量热线：(010)81055316
反盗版热线：(010)81055315

前　言

关于设计模式

模式是从不断重复出现的事件中发现和抽象出的规律，是解决问题形成的经验总结。设计模式作为一种模式，最早应用于建筑领域，目的是在图纸上以一种结构化、可重用化的方法，获得建筑的基本要素。渐渐地，这种思想在软件领域流行起来，并获得发展，形成了软件开发的设计模式。

软件设计模式被认为是一套被反复使用、多数人知晓、经过分类编目的代码设计经验的总结。最早的设计模式是由 GOF 在《Design Patterns: Elements of Reusable Object-Oriented Software》一书提出的，这也被称为经典设计模式，共有 23 个，分为创建型模式、行为型模式、结构型模式三类。使用设计模式的目的是为了提高代码的可重用性、让代码更容易被他人理解、系统更加可靠。

创作背景

应用设计模式构建有弹性、可扩展的应用系统已成为软件人员的共识，越来越多的程序员需要掌握设计模式的内容。近年来，市场上也涌现了一些有关设计模式的书籍。这些书籍各有特点，多从生活中的示例入手，让读者对所述设计模式有一定的感性认识，然后引入设计模式概念，最后用计算机专业程序进行理性说明。通常，示例部分内容成熟，但专业应用部分的讲解稍显单薄，笔者认为主要有以下几点。第一，示例偏简单，读者看过一遍就理解其含义，但很难会真正应用；第二，示例趣味性不足，例如，很多讲解基于命令行界面，而实际应用更多的是图形界面；第三，有些书以 ERP 各个具体模块讲解设计模式，显得过于单一。上述原因是促使我们写作本书的动力。

本书内容

本书首先利用两章讲解了用到的预备知识：接口与抽象类，反射。然后从常用的 23 个设计模式中精选了 10 个进行讲解，包括 2 个创建型模式：工厂、生成器模式，4 个行为型模式：观察者、访问者、状态、命令模式，4 个结构型模式：桥接、代理、装饰器、组合模式。每个模式一般都包含以下四部分。

（1）问题的提出：一般从生活中的一类常见事物引出待讨论的主题。

（2）模式讲解：用模式方法解决与之对应的最基本问题，归纳出角色及 UML 类图。

（3）深入理解模式：讲解笔者对模式的一些体会。

（4）应用探究：均是实际应用中较难的程序，进行了详细的问题分解、分析与说明。

本书特色

（1）示例丰富，讲解细致，有命令行程序，也有图形界面、Web 程序等，涉及 Java、JSP、JavaScript、Ajax 等技术。

（2）强调了语义的作用。一方面把设计模式抽象转化成日常生活中最朴实的语言；另一方面把生活中对某事物“管理”的语言转译成某设计模式。相比而言，后者更为重要。

（3）强调了反射技术的作用。对与反射技术相关的设计模式均做了详细的论述。

（4）提出了如何用接口思维巧妙实现 C++标准模板库方法功能的技术手段。

学习设计模式方法

（1）在清晰设计模式基础知识的基础上，认真实践应用探究中的每一个示例，并充分分析，加以思考。

（2）学习设计模式不是一朝一夕的事，不能好高骛远。它是随着读者思维的发展而发展的，一定要在项目中亲身实践，量变引起质变，有句话说得好：“纸上得来终觉浅，决知此事要躬行”。

（3）加强基础知识训练，如数据结构、常用算法等。基础知识牢固了，学习任何新事物都不会发慌，有信心战胜它。否则，知识学得再多，也只是空中楼阁。

（4）不要为了模式而模式，要在项目中综合考虑，统筹安排。

关于本书的使用

本书提供各章节的完整代码供读者使用。

（1）由于篇幅关系，大多数程序的导入（import）命令在书中的示例中没有给出，这些语句需要读者自行加入。但给出的程序源码是完整的（包含导入命令）。

（2）在使用反射时，例如，使用 XML 文件或 properties 文件封装类的配置信息时，如果被封装的类在一个包中，应在配置文件中类的名字之前指明该类所属的包，这样程序才能顺利编译。

（3）Web 程序使用 Tomcat 服务器，版本是 Tomcat 7.0。

（4）部分章节需要连接数据库，本书的数据库采用 MySQL 5.5。数据库操作需要 MySQL 驱动程序，本书使用的是 connection-java-5.1.17-bin.jar，可从 MySQL 官网（http://dev.mysql.com/downloads/ connector/）下载。之后需要添加到项目的 classpath 环境变量中；如果是 Java Web 应用，应将驱动程序复制到 Web 项目的 WEB-INF/lib 文件夹中。

（5）连接数据库的工具类 DbProc 位于第 4 章，全书通用。如果需要，复制到相应的项目中即可。

（6）本书使用的数据库名字是 test，包括 login、student、teacher、score 等表。建立表的 SQL 命令在源码的文件 create.txt 中。

总之，设计模式是一门重要的计算机软件开发技术，笔者希望尽一些微薄之力，为我国的设计模式研究添砖加瓦。但由于水平有限，时间紧迫，书中难免有不妥或疏漏之处，恳请广大读者批评指正。

编　者

2014 年 3 月

目　录

第 1 章　接口与抽象类

1.1　语义简单描述

接口与抽象类是面向对象思想的两个重要概念。接口仅是方法定义和常量值定义的集合，方法没有函数体；抽象类定义的内容理论上比接口中的内容要多得多，可定义普通类所包含的所有内容，还可定义抽象方法，这也正是叫作抽象类的原因所在。接口、抽象类本身不能示例化，必须在相应子类中实现抽象方法，才能获得应用。

那么，如何更好地理解接口与抽象类？接口中能定义抽象方法，为什么还要抽象类？抽象方法无函数体，不能示例化，说明接口、抽象类本身没有用途，这种定义有意义吗？接口与抽象类的关系到底如何？

获得这些问题答案的最好办法就是来自于生活实践。例如写作文的时候，一定要先思考好先写什么，后写什么；做几何题的时候，要想清楚如何引辅助线，用到哪些公理、定理等；做科研工作的时候，一定要思索哪些关键问题必须解决；工厂生产产品前，必须制订完善的生产计划等。也就是说，人们在做任何事情前，一般来说是先想好，再去实现。这种模式在生活中是司空见惯的。因此，Java 语言一定要反映“思考 - 实现”这一过程，通过不同关键字来实现，即用接口（interface）、抽象类（abstract）来反映思考阶段，用子类（class）来反映实现阶段。

从上文易于得出：“思考 - 实现”是接口、抽象类的简单语义。从此观点出发，结合生活实际，可以方便回答许多待解答的问题，具体如下。

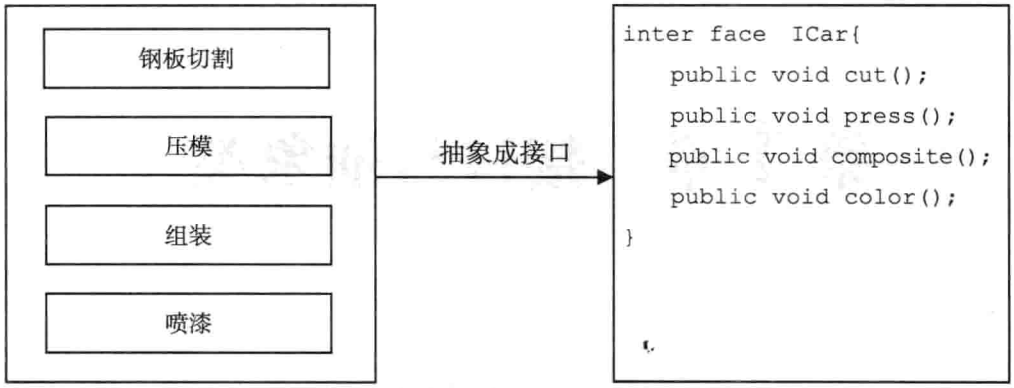
为什么接口、抽象类不能示例化？由于接口、抽象类是思考的结果，只是提出了哪些问题需要解决，无需函数体具体内容，当然不能示例化了。

为什么接口、抽象类必须由子类实现？提出问题之后，总得有解决的方法。Java 语言是通过子类来实现的，当然一定要解决“思考”过程中提出的所有问题。

接口、抽象类有什么区别？可以这样考虑，人类经思考后提出的问题一般有两类：一类是“顺序”问题，另一类是“顺序+共享”问题。前者是用接口描述的，后者是用抽象类来描述的。

图 1-1 所示为一个生产小汽车具体的接口示例。

图 1-1（a）所示为生产小汽车可由钢板切割、压模、组装、喷漆 4 个工序组成。这些工序是顺序关系，因此转化成接口是最恰当的，如图 1-1（b）所示。



（a）小汽车生产抽象工序　　（b）转化成的接口

图 1-1　生产小汽车接口示例

抽象类与接口不同，假设要组装多种价位的电脑，其配置参数如图 1-2 所示。

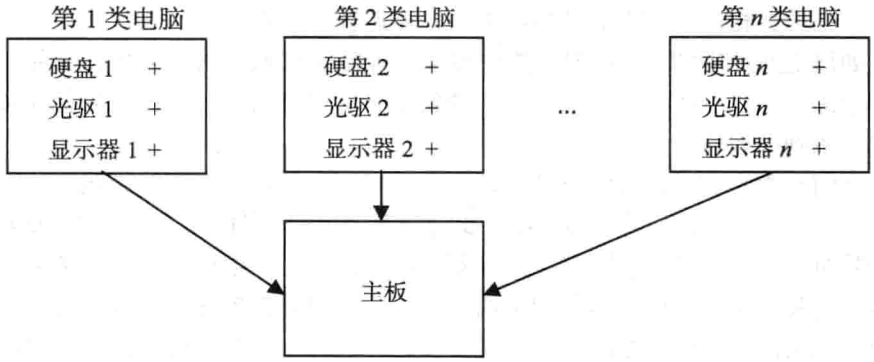


图 1-2　电脑抽象类示例

可以看出，要配置 n 种电脑。每种电脑的硬盘、光驱、显示器是不同类型的，属于并列结构（也可以看作顺序结构）；主板是相同类型的，属于共享结构。因此，转化成的抽象类如下。

```
abstract class Computer{
    abstract void makeHarddisk();        //表明每类电脑有不同的硬盘
    abstract void makeOptical();         //表明每类电脑有不同的光驱
    abstract void makeMonitor();         //表明每类电脑有不同的显示器
    void makeMainBoard(){ };             //表明所有类型的电脑有相同类型的主板
}
```

1.2　与框架的关系

通过 1.1 节的描述可以看出，接口、抽象类代表了提出的两类问题的抽象字符描述。这是理解接口、抽象类的基础，是编制框架的关键。框架包括方法框架与流程框架。下面通过

示例加以说明。

【例 1-1】 方法框架示例。编制求对象数组最大值的泛型方法。

```
//ILess.java:定义二元比较方法
public interface ILess<T> {
    boolean less(T x, T y);
}

//Algo.java:泛型方法类
public class Algo<T> {
    public T getMax(T t[], ILess<T> cmp){
        T maxValue = t[0];
        for(int i=1; i<t.length; i++){
            if(cmp.less(maxValue, t[i]))        //这一行是理解的关键
                maxValue = t[i];
        }
        return maxValue;
    }
}
```

（1）求对象数组的最大值算法比较简单，这里就不多言了。ILess 接口定义了二元比较方法，getMax()是求对象数组最大值的泛型方法。可以发现，根本无须实现 ILess 的子类，上述框架程序即编译成功。

（2）框架程序若获得具体应用，则必须实现 ILess 的子类。以求整型数组最大值及学生成绩最大值加以说明，代码如下。

```
//InteLess.java: 整型数比较器
public class InteLess implements ILess<Integer> {
    public boolean less(Integer x, Integer y) {
        return x<y;
    }
}

//Student.java: 学生基本类
public class Student {
    String name; //姓名
    int grade;   //成绩
    public Student(String name, int grade){
        this.name = name;
        this.grade= grade;
    }
}

//StudLess.java: 学生成绩比较器
public class StudLess implements ILess<Student> {
    public boolean less(Student x, Student y) {
        return x.grade < y.grade;
    }
}

//Test.java: 测试类
```

```
public class Test {
    public static void main(String[] args ) {
        Algo<Integer> obj = new Algo();
        ILess<Integer> cmp = new InteLess();
        Integer a[] = {3,9,2,8};
        Integer max = obj.getMax(a, cmp);
        System.out.println("Integer max=" +max);

        Algo<Student> obj2 = new Algo();
        ILess<Student> cmp2 = new StudLess();
        Student s[]={new Student("li",70),new Student("sun",90),
                     new Student("zhao",80)};
        Student max2 = obj2.getMax(s, cmp2);
        System.out.println("Student max grade:" + max2.grade);
    }
}
```

（3）根据上文可以看出，要实现求某类对象数组的最大值，主要工作是编制具体的从 ILess 接口派生的子类代码，重写 less()方法，自定义比较规则即可。

【例 1-2】 流程框架示例：求圆、矩形的面积。要求：当求圆面积时，能输入圆的半径；当求矩形面积时，能输入长、宽。

分析：根据题目特征得出，对不同的形状有不同的输入参数，有不同的求面积算法。“输入”及“求面积”功能是并列关系，因此，用接口来定义“输入”及“求面积”功能是最恰当的。由此接口出发，得到的相关类代码如下。

```
//IShape.java : 形状接口定义
public interface IShape {
    boolean input();                    //输入方法
    float getArea();                    //求面积方法
}

//ShapeProc.java: 流程处理类
public class ShapeProc {
    private IShape shape;
    public ShapeProc(IShape shape){
        this.shape = shape;
    }

    public float process(){             //每个形状处理包括输入及求面积两步
        shape.input();                  //输入功能
        float value = shape.getArea(); //求面积功能
        return value;                   //返回面积
    }
}
```

（1）ShapeProc 是对接口多态对象的封装类。process()方法表明了对某形状的统一处理过程，包括参数输入 input()方法及求面积 getArea()方法。可以发现，根本无需实现 IShape 的子类，上述流程框架程序即编译成功。

（2）流程框架程序若获得具体应用，则必须实现 IShape 的子类，圆类、矩形类及测试类代码如下。

```
//Circle.java:圆类
import java.util.*;
public class Circle implements IShape {
    float r;
    public float getArea() {
        float s = (float)Math.PI*r*r;
        return s;
    }
    public boolean input() {
        System.out.println("请输入半径:");
        Scanner s = new Scanner(System.in);
        r = s.nextFloat();
        return true;
    }
}
//Rect.java:矩形类
import java.util.*;
public class Rect implements IShape {
    float width,height;
    public float getArea() {
        float s = width*height;
        return s;
    }
    public boolean input() {
        System.out.println("请输入宽、高:");
        Scanner s = new Scanner(System.in);
        width = s.nextFloat();
        height = s.nextFloat();
        return true;
    }
}
//Test.java:测试类
public class Test{
    public static void main(String[] args) {
        IShape shape = new Circle();
        ShapeProc obj = new ShapeProc(shape);
        float value = obj.process();
        System.out.println("圆面积:" + value);

        IShape shape2 = new Rect();
        ShapeProc obj2 = new ShapeProc(shape);
        float value2 = obj.process();
        System.out.println("矩形面积:" + value);
    }
}
```

（3）根据上文可以看出，要实现求某形状的面积，主要工作是编制具体的从 IShape 接口派生的子类代码，重写 input()、getArea()方法，而流程代码无需重写，共享在 ShapeProc 类中的 process()方法中了。

通过例 1-1、例 1-2 可以得出一个重要的框架编程原则：面向接口、抽象类进行编程。也就是说，只要接口、抽象类是稳定的，一般可以抛开具体的实现子类进行编程，容易形成一个稳定的框架系统。

1.3 拓 展 研 究

1.3.1 柔性多态

1. 问题提出

仍以求圆和长方形面积为例。假设其类图如图 1-3 所示。

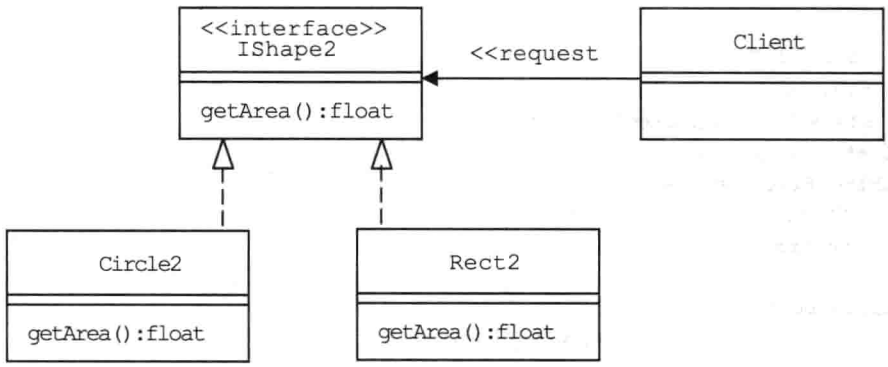


图 1-3 圆和长方形面积功能类图

可以看出这是常规的多态程序设计，父类是 IShape，多态接口函数是 float getArea()；子类 Circle2 及 Rectangle 分别重写了多态函数 getArea()；客户端通过动态绑定对接口编程实现了求圆或长方形面积的功能。

但是如果随着时间的推移还需要求圆和长方形的周长，该如何修改程序呢？

普通的思路是：重新定义接口 IShape2，增加求周长接口函数 float getPerimeter()， 再在 Circle2 及 Rect2 类中实现 getPerimeter()函数的具体功能。这势必造成接口及实现模块、客户端程序都需要修改并重新编译。这是我们不希望看到的，我们希望仅底层具体模块功能可以修改并编译，而接口、上层模块及客户端程序不要重新编写。

因此，如何巧妙运用多态满足不断变化需求分析的需要，只修改需要改变的具体模块，其他模块不修改，是即将论述的中心内容。

普通多态编程局限性：如果接口函数内容发生变化，那么相应的各实现子类必须发生变化，导致相关联的各级模块必须重新编程及编译，这即是普通多态编程的局限性。造成这一结果的主要原因是父类、子类定义的多态函数关联过强，消除这种关联性是实现柔性多态功能的关键。

2. 柔性多态代码示例

柔性多态是指程序架构必须满足不断发展的需求分析的需要，只需修改需要改变的子模块，而相关联模块及主程序都不需要变化。以求圆和长方形面积、周长为例，采用柔性多态，

具体代码如下。

```
//IShape2.java:定义柔性多态接口
public interface  IShape2 {
    //多态函数定义
    public Object dispatch(int nID,Object in);
}

//Circle2.java:圆类
class Circle2  implements IShape2
{
    public float r;
    public Circle2(float r){
        this.r = r;
    }
//多态方法
    public Object dispatch(int nID,Object in){
        Object obj=null;
        switch(nID){
        case 0:
            obj = getArea(in);break;
        case 1:
            obj = getPerimeter(in);break;
        }
        return obj;
    }
Object getArea(Object in){                    //非多态方法
        float area = (float)Math.PI*r*r;
        return new Float(area);
    }
    Object getPerimeter(Object in){           //非多态方法
        float len = (float)Math.PI*r*2.0f;
        return new Float(len);
    }
}
```

从上述代码可以得出柔性多态的设计思想，具体如下。

- 接口内容固定，如 IShape2 中仅定义了一个多态接口方法 dispatch()。
- 子类中重写的多态函数 dispatch 仅起到转发作用，且转发的具体函数都不是多态函数，如 Circle2 类中的 getArea()、getPerimeter()都只是普通函数，这正和普通多态接口编程思想不一致。

例如，如果现在增加一个功能：求圆内接正三角形边长。可以仅在 Circle2 类中增加一个普通方法 getTriangleLen()，再在多态方法 dispatch()中增加一个 case 开关，调用 getTriangleLen()方法就可以了。而对接口 IShape2 根本就没有修改。

从中可以看出，接口定义的多态方法 dispatch()与子类中的各普通具体方法间的关系是“间接的”，不是“直接的”，是转发关系，削弱了父子类多态方法的强关联，是实现柔性多态的关键。

3. 对 dispatch()方法参数的理解

该方法有两个参数：各具体普通函数的功能号 nID，是一个整型数，在同一模块中不能

有重复值；输入参数 in，类型是 Object，相当于泛型编程，使程序灵活，一般不要定义成具体的值或类类型。

该函数返回值是 Object 对象。若为 null，表明计算失败；若非空，则在调用方用强制类型转换才能得到所需要的结果。

一个简单测试类代码如下。

```
public class Test {
    public static void main(String []args){
        IShape2 obj = new Circle2(10.0f);
        Float result = (Float)obj.dispatch(1,null);
       System.out.println("半径 10 圆面积:"+result.floatValue());
    }
}
```

4. 进一步完善

可以看出，要完成所需功能，必须知道相应具体函数的转发 ID 号。由于 ID 号是一个整型数，表意不明显，按人的思维角度不容易记忆，按字符串记忆更符合常规习惯。因此，完善后的模块应有以下主要功能：多态模块中应该给各个具体函数赋有意义的特征字符串值；调用端通过特征字符串值查询具体函数的 ID 号；根据 ID 号执行具体的转发函数。接口定义完善如下所示。也就是说，增加了一个多态函数 query，功能是查询特征字符串对应的功能 ID 号，若没有查询到，则返回-1。

```
interface IShape2{
    public int query(String strID);
    public Object dispatch(int nID,Object in);
}
```

以 Circle 类为例，完善后代码（仅列出不同部分）如下。

```
class Circle2 implements IShape2 {
    static Vector<String> vec = new Vector();
    static {
        vec.add("getArea");
        vec.add("getPermeter");
    }
    public int query(String strID){
        int nID = vec.indexOf(strID);
        return nID;
    }
    ..... //其余略
}
```

相应的测试类代码如下所示。

```
public class Test {
    public static void main(String []args){
    Shape obj = new Circle2(10.0f);
    int nID = obj.query("getArea");
    Float result = (Float)obj.dispatch(nID,null);
    System.out.println("半径 10 圆面积:"+result.floatValue());
    }
}
```

5. 总结

固化父类接口函数定义，子类通过重写多态派发函数，是柔性多态的基本设计思想。

本例中 IShape2 接口是各种形状子类的父类，其实它还可以做其他任何需要柔性多态模块的共同父类。因此接口名可取得更一般些，例如接口定义如下。

```
interface Flexible_Interface{
    public int query(String strID);
    public Object dispatch(int nID,Object in);
}
```

1.3.2　借鉴 STL 标准模板库

1. 基本思路

STL（Standard Template Library）是标准模板库的简称，属于 C++知识体系。与 Java 语言相比，STL 的优势是它有 100 个左右的泛型方法，涵盖了绝大多数常用算法，而 JDK 中仅有全排序、二分查找等少量算法。因此，把 STL 中的泛型方法代码移植到 Java 中是一个较好的开发思路。

STL 能实现泛型的一个重要因素是 C++支持操作符重载，主要是 operator==、operator< 两个二元操作符。Java 语言可以用接口来替换，假设为 IComparator 比较器接口，为了使用方便，抽象类 AbstractComparator 定义了该接口的一个默认实现。代码如下。

```
//IComparator.java
public interface IComparator<T> {
    boolean equal(T x, T y);
    boolean less(T x, T y);
}

//AbstractComparator.java
public class AbstractComparator<T> implements IComparator<T> {
    public boolean equal(T x, T y) {
        return true;
    }
    public boolean less(T x, T y) {
        return true;
    }
}
```

2. 典型示例

【例 1-3】 编制三个对象求中值的算法。

为了说明问题，列出了采自 VC 6.0 的 STL 中该算法源码，具体如下。

```
template<class _Ty> inline
_Ty _Median(_Ty _X, _Ty _Y, _Ty _Z){
if (_X < _Y)
        return (_Y < _Z ? _Y : _X < _Z ? _Z : _X);
    else
        return (_X < _Z ? _X : _Y < _Z ? _Z : _Y);
}
```

转换后的 Java 代码如下。

```
public class Algorithm<T> {
    IComparator<T> cmp;   //比较器，是 AbstractComparator 的子类
```

```
    Algorithm (IComparator<T> cmp){
        this.cmp = cmp;   //初始化比较器
    }
    public  T median(T x, T y, T z){
        if(cmp.less(x, y))
            return cmp.less(y, z)?y:cmp.less(x, z)?z:x;
        else
            return cmp.less(x, z)?x:cmp.less(y, z)?z:y;
    }
}
```

- 由于 IComparator 接口对象为算法类 Algorithm 中所有方法共享，所以把它定义为成员变量，并在构造方法中加以初始化。
- 可以看出，C++代码与 Java 代码大同小异，主要把“<”用“less()”方法进行替换。STL 中有许多方法可以用类似这样简单的替换，而无需考虑各种细节（如边界条件等），就能直接转化成 Java 代码。得出的代码是专家级的，稳定性强。

【例 1-4】 局部排序。

STL 中排序主要包括全排序 sort()方法、局部排序 partial_sort()方法、第 nth 排序 nth_element()方法，而 JDK 中仅有全排序 sort()方法。因此，编制稳定的局部排序、求第 nth 元素排序是必要的。借鉴 STL，可以很快编制所需排序程序，以局部排序 partial_sort()为例，代码如下（在上文中的类 Algorithm 中添加）。

```
public class Algorithm<T> {
    //……略去代码同例 1-3
    public boolean push_heap(T[] t, int h, int j, T v){
        for(int i=(h-1)/2; j<h && cmp.less(t[i], v); i=(h-1)/2){
            t[h] = t[i];
            h = i;
        }
        t[h] = v;
        return true;
    }

    public boolean pop_heap(T[] t, int m, int i){
        t[i] = t[0];
        adjust_heap(t, 0, m);
        return true;
    }

    public boolean sort_heap(T[] t, int l){
        for(; l>1; --l){
            pop_heap(t, l-1, l-1);
        }
        return true;
    }

    public boolean adjust_heap(T[] t, int pos, int nSize){
        int j=pos;
        T v = t[pos];
        int k = 2*pos+2;
        for(; k<nSize; k=2*k+2 ){
```