java设计模式之享元模式

当前咱们国家正在大力倡导构建和谐社会，其中一个很重要的组成部分就是建设资源节约型社会，“浪费可耻，节俭光荣”。在软件系统中，有时候也会存在资源浪费的情况，例如在计算机内存中存储了多个完全相同或者非常相似的对象，如果这些对象的数量太多将导致系统运行代价过高，内存属于计算机的“稀缺资源”，不应该用来“随便浪费”，那么是否存在一种技术可以用于节约内存使用空间，实现对这些相同或者相似对象的共享访问呢？答案是肯定，这种技术就是我们本章将要学习的享元模式。

14.1 围棋棋子的设计

Sunny软件公司欲开发一个围棋软件，其界面效果如图14-1所示：

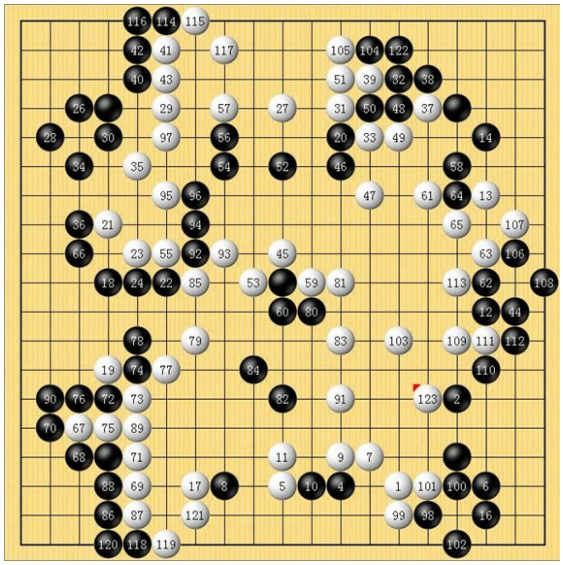

图14-1 围棋软件界面效果图

Sunny软件公司开发人员通过对围棋软件进行分析，发现在围棋棋盘中包含大量的黑子和白子，它们的形状、大小都一模一样，只是出现的位置不同而已。如果将每一个棋子都作为一个独立的对象存储在内存中，将导致该围棋软件在运行时所需内存空间较大，**如何降低运行代价、提高系统性能是Sunny公司开发人员需要解决的一个问题**。为了解决这个问题，Sunny公司开发人员决定使用享元模式来设计该围棋软件的棋子对象，那么享元模式是如何实现节约内存进而提高系统性能的呢？别着急，下面让我们正式进入享元模式的学习。

14.2 享元模式概述

当一个软件系统在运行时产生的对象数量太多，将导致运行代价过高，带来系统性能下降等问题。例如在一个文本字符串中存在很多重复的字符，如果每一个字符都用一个单独的对象来表示，将会占用较多的内存空间，那么我们如何去避免系统中出现大量相同或相似的对象，同时又不影响客户端程序通过面向对象的方式对这些对象进行操作？享元模式正为解决这一类问题而诞生。享元模式通过共享技术实现相同或相似对象的重用，**在逻辑上每一个出现的字符都有一个对象与之对应，然而在物理上它们却共享同一个享元对象**，这个对象可以出现在一个字符串的不同地方，相同的字符对象都指向同一个实例，在享元模式中，存储这些共享实例对象的地方称为**享元池(Flyweight Pool)**。我们可以针对每一个不同的字符创建一个享元对象，将其放在享元池中，需要时再从享元池取出。如图14-2所示：

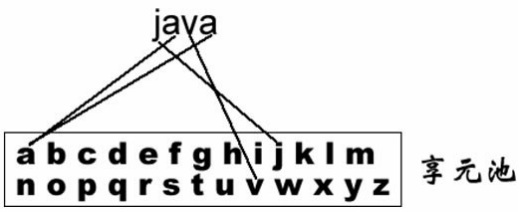

图14-2 字符享元对象示意图

享元模式以共享的方式高效地支持大量细粒度对象的重用，享元对象能做到共享的关键是区分了**内部状态(Intrinsic State)**和**外部状态(Extrinsic State)**。下面将对享元的内部状态和外部状态进行简单的介绍：

（1）**内部状态是存储在享元对象内部并且不会随环境改变而改变的状态，内部状态可以共享**。如字符的内容，不会随外部环境的变化而变化，无论在任何环境下字符“a”始终是“a”，都不会变成“b”。

(2) **外部状态是随环境改变而改变的、不可以共享的状态。**享元对象的外部状态通常由客户端保存，并在享元对象被创建之后，需要使用的时候再传入到享元对象内部。一个外部状态与另一个外部状态之间是相互独立的。如字符的颜色，可以在不同的地方有不同的颜色，例如有的"a"是红色的，有的"a"是绿色的，字符的大小也是如此，有的"a"是五号字，有的"a"是四号字。而且字符的颜色和大小是两个独立的外部状态，它们可以独立变化，相互之间没有影响，客户端可以在使用时将外部状态注入享元对象中。

正因为区分了内部状态和外部状态，我们可以**将具有相同内部状态的对象存储在享元池中**，享元池中的对象是可以实现共享的，需要的时候就将对象从享元池中取出，实现对象的复用。通过向取出的对象注入不同的外部状态，可以得到一系列相似的对象，而这些对象在内存中实际上只存储一份。

享元模式定义如下：

享元模式(Flyweight Pattern)：运用共享技术有效地支持大量细粒度对象的复用。系统只使用少量的对象，而这些对象都很相似，状态变化很小，可以实现对象的多次复用。由于享元模式要求能够共享的对象必须是细粒度对象，因此它又称为轻量级模式，它是一种对象结构型模式。

享元模式结构较为复杂，一般结合工厂模式一起使用，在它的结构图中包含了一个享元工厂类，其结构图如图14-3所示：

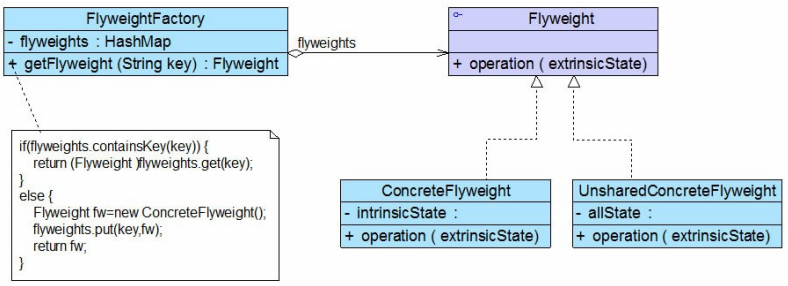

图14-3 享元模式结构图

在享元模式结构图中包含如下几个角色：

● Flyweight(抽象享元类)：通常是一个接口或抽象类，在抽象享元类中声明了具体享元类公共的方法，这些方法可以向外界提供享元对象的内部数据(内部状态)，同时也可以通过这些方法来设置外部数据(外部状态)。

● ConcreteFlyweight(具体享元类)：它实现了抽象享元类，其实例称为享元对象；在具体享元类中为内部状态提供了存储空间。通常我们可以结合单例模式来设计具体享元类，为每一个具体享元类提供唯一的享元对象。

● UnsharedConcreteFlyweight(非共享具体享元类)：并不是所有的抽象享元类的子类都需要被共享，不能被共享的子类可设计为非共享具体享元类；当需要一个非共享具体享元类的对象时可以直接通过实例化创建。

● FlyweightFactory(享元工厂类)：享元工厂类用于创建并管理享元对象，它针对抽象享元类编程，将各种类型的具体享元对象存储在一个享元池中，享元池一般设计为一个存储"键值对"的集合(也可以是其他类型的集合)，可以结合工厂模式进行设计；当用户请求一个具体享元对象时，享元工厂提供一个存储在享元池中已创建的实例或者创建一个新的实例(如果不存在的话)，返回新创建的实例并将其存储在享元池中。

在享元模式中引入了享元工厂类，享元工厂类的作用在于提供一个用于存储享元对象的享元池，当用户需要对象时，首先从享元池中获取，如果享元池中不存在，则创建一个新的享元对象返回给用户，并在享元池中保存该新增对象。典型的享元工厂类的代码如下：

```
class FlyweightFactory {
   //定义一个HashMap用于存储享元对象，实现享元池
     private HashMap flyweights = newHashMap();


     public Flyweight getFlyweight(String key){
          //如果对象存在，则直接从享元池获取
          if(flyweights.containsKey(key)){
               return(Flyweight)flyweights.get(key);
          }
          //如果对象不存在，先创建一个新的对象添加到享元池中，然后返回
          else {
               Flyweight fw = newConcreteFlyweight();
               flyweights.put(key,fw);
               return fw;
          }
     }
}
```

享元类的设计是享元模式的要点之一，在享元类中要将内部状态和外部状态分开处理，通常将内部状态作为享元类的成员变量，而外部状态通过注入的方式添加到享元类中。典型的享元类代码如下所示：

```
class Flyweight {
    //内部状态intrinsicState作为成员变量，同一个享元对象其内部状态是一致的
     private String intrinsicState;
```

```
    public  Flyweight(String intrinsicState) {
        this.intrinsicState=intrinsicState;
    }

    //外部状态extrinsicState在使用时由外部设置，不保存在享元对象中，即使是同一个对象，在每一次调用时也可以传入不同的外部状态
    public void operation(String  extrinsicState) {
        ......
    }
}
```

14.3 完整解决方案

为了节约存储空间，提高系统性能，Sunny公司开发人员使用享元模式来设计围棋软件中的棋子，其基本结构如图14-4所示：

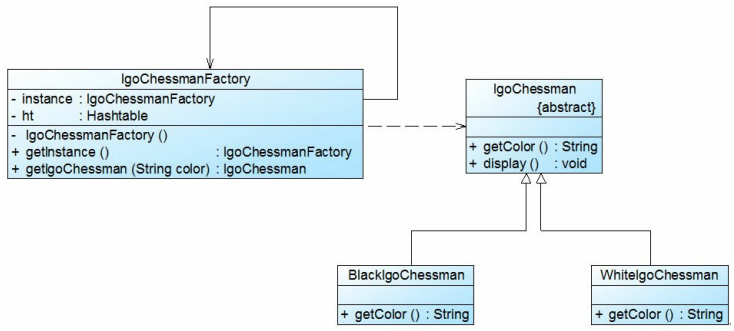

图14-4 围棋棋子结构图

在图14-4中，IgoChessman充当抽象享元类，BlackIgoChessman和WhiteIgoChessman充当具体享元类，IgoChessmanFactory充当享元工厂类。完整代码如下所示：

```
import java.util.*;

//围棋棋子类:抽象享元类
abstract class IgoChessman {
    public abstract String getColor();

    public void display() {
        System.out.println("棋子颜色:" + this.getColor());
    }
}

//黑色棋子类:具体享元类
class BlackIgoChessman extends IgoChessman {
    public String getColor() {
        return "黑色";
    }
}

//白色棋子类:具体享元类
class WhiteIgoChessman extends IgoChessman {
    public String getColor() {
        return "白色";
    }
}

//围棋棋子工厂类:享元工厂类，使用单例模式进行设计
class IgoChessmanFactory {
    private static IgoChessmanFactory instance = new IgoChessmanFactory();
    private static Hashtable ht; //使用Hashtable来存储享元对象，充当享元池

    private IgoChessmanFactory() {
        ht = new Hashtable();
        IgoChessman black,white;
        black = new BlackIgoChessman();
        ht.put("b",black);
        white = new WhiteIgoChessman();
        ht.put("w",white);
    }

    //返回享元工厂类的唯一实例
    public static IgoChessmanFactory getInstance() {
        return instance;
    }

    //通过key来获取存储在Hashtable中的享元对象
    public static IgoChessman getIgoChessman(String color) {
        return (IgoChessman)ht.get(color);
```

```
    }
}
```

编写如下客户端测试代码：

```
class Client {
    public static void main(String args[]) {
        IgoChessman black1,black2,black3,white1,white2;
        IgoChessmanFactory factory;

        //获取享元工厂对象
        factory = IgoChessmanFactory.getInstance();

        //通过享元工厂获取三颗黑子
        black1 = factory.getIgoChessman("b");
        black2 = factory.getIgoChessman("b");
        black3 = factory.getIgoChessman("b");
        System.out.println("判断两颗黑子是否相同:" + (black1==black2));

        //通过享元工厂获取两颗白子
        white1 = factory.getIgoChessman("w");
        white2 = factory.getIgoChessman("w");
        System.out.println("判断两颗白子是否相同:" + (white1==white2));

        //显示棋子
        black1.display();
        black2.display();
        black3.display();
        white1.display();
        white2.display();
    }
}
```

编译并运行程序，输出结果如下：

```
判断两颗黑子是否相同:true
判断两颗白子是否相同:true
棋子颜色:黑色
棋子颜色:黑色
棋子颜色:黑色
棋子颜色:白色
棋子颜色:白色
```

从输出结果可以看出，虽然我们获取了三个黑子对象和两个白子对象，但是它们的内存地址相同，也就是说，它们实际上是同一个对象。在实现享元工厂类时我们使用了单例模式和简单工厂模式，确保了享元工厂对象的唯一性，并提供工厂方法来向客户端返回享元对象。

14.5 带外部状态的解决方案

Sunny软件公司开发人员通过对围棋棋子进行进一步分析，发现虽然黑色棋子和白色棋子可以共享，但是它们将显示在棋盘的不同位置，如何让相同的黑子或者白子能够多次重复显示且位于一个棋盘的不同地方？解决方法就是将棋子的位置定义为棋子的一个外部状态，在需要时再进行设置。因此，我们在图14-4中增加了一个新的类Coordinates(坐标类)，用于存储每一个棋子的位置，修改之后的结构图如图14-5所示：

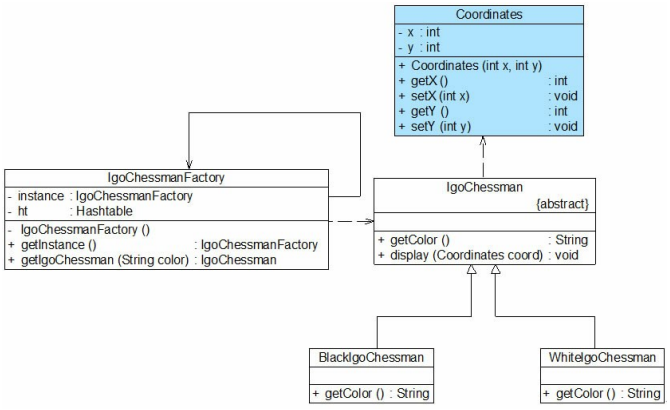

图14-5 引入外部状态之后的围棋棋子结构图

在图14-5中，除了增加一个坐标类Coordinates以外，抽象享元类IgoChessman中的display()方法也将对应增加一个Coordinates类型的参数，用于在显示棋子时指定其坐标，Coordinates类和修改之后的IgoChessman类的代码如下所示：

```
class Coordinates {
    private int x;
    private int y;

    public Coordinates(int x,int y) {
```

```
        this.x = x;
        this.y = y;
    }

    public int getX() {
        return this.x;
    }

    public void setX(int x) {
        this.x = x;
    }

    public int getY() {
        return this.y;
    }

    public void setY(int y) {
        this.y = y;
    }
}

//围棋棋子类:抽象享元类
abstract class IgoChessman {
    public abstract String getColor();

    public void display(Coordinates coord){
        System.out.println("棋子颜色:" + this.getColor() + ", 棋子位置:" + coord.getX() + ", " + coord.getY() );
    }
}
```

客户端测试代码修改如下:

```
class Client {
    public static void main(String args[]) {
        IgoChessman black1,black2,black3,white1,white2;
        IgoChessmanFactory factory;

        //获取享元工厂对象
        factory = IgoChessmanFactory.getInstance();

        //通过享元工厂获取三颗黑子
        black1 = factory.getIgoChessman("b");
        black2 = factory.getIgoChessman("b");
        black3 = factory.getIgoChessman("b");
        System.out.println("判断两颗黑子是否相同:" + (black1==black2));

        //通过享元工厂获取两颗白子
        white1 = factory.getIgoChessman("w");
        white2 = factory.getIgoChessman("w");
        System.out.println("判断两颗白子是否相同:" + (white1==white2));

        //显示棋子,同时设置棋子的坐标位置
        black1.display(new Coordinates(1,2));
        black2.display(new Coordinates(3,4));
        black3.display(new Coordinates(1,3));
        white1.display(new Coordinates(2,5));
        white2.display(new Coordinates(2,4));
    }
}
```

编译并运行程序,输出结果如下:

```
判断两颗黑子是否相同:true
判断两颗白子是否相同:true
棋子颜色:黑色, 棋子位置:1, 2
棋子颜色:黑色, 棋子位置:3, 4
棋子颜色:黑色, 棋子位置:1, 3
棋子颜色:白色, 棋子位置:2, 5
棋子颜色:白色, 棋子位置:2, 4
```

从输出结果可以看到,在每次调用display()方法时,都设置了不同的外部状态——坐标值,因此相同的棋子对象虽然具有相同的颜色,但是它们的坐标值不同,将显示在棋盘的不同位置。

【作者:刘伟 http://blog.csdn.net/lovelion】

分类: Java


关注 - 17
粉丝 - 51
+加关注

3 推荐　0 反对

« 上一篇：java设计模式之外观模式

» 下一篇：java设计模式之代理模式

posted @ 2017-05-06 16:19 持&恒 阅读(2557) 评论(1) 编辑 收藏

评论列表

#1楼 2018-05-02 17:07 苍狼小跟班

受益匪浅，写的很好！

支持(1) 反对(0)

刷新评论 刷新页面 返回顶部

注册用户登录后才能发表评论，请 登录 或 注册，访问网站首页。

相关博文：
· Java设计模式之享元模式
· 设计模式 之 享元模式
· 设计模式之享元模式
· java设计模式--享元模式
· JAVA 设计模式 享元模式

热门推荐：
· HTML中<input>和<textarea>的区别
· <input type="text">和<textarea>的区别
· “暗网”真的如传言般可怕吗？这是一份《暗网指南》
· Pycharm安装第三方库
· webstorm 2018激活码