java苹果+番茄的博客文章

作者: java苹果+番茄 http://liulve-rover-163-com.javaeye.com

我的博客文章精选

目 录

1. java设计与模式

1.1 java设计模式笔记【创建模式第一篇】

发表时间: 2009-10-22

前序：现在开始将自己整理的一些java设计模式的笔记与大家共享，希望大家不吝指正，谢谢。
声明：虽然这些都是出自《java设计与模式》一书中，但是是本人的学习心得笔记，请尊重本人劳动，谢谢

抽象工厂（Abstract Factory）模式：

1、抽象工厂模式可以向客户端提供一个接口，使得客户端在不必指定产品的具体类型的情况下，创建多个产品类中的产品对象，这就是抽象工厂模式的用意。

2、对抽象工厂模式的用意的理解：
第一段：一个系统需要多个抽象产品角色，这些抽象产品角色可以用java类实现。
第二段：根据里氏原则，任何接受父类型的地方，都应当能够接受子类型。因此，实际上系统所需要的，仅仅是类型与这些抽象产品角色形同的一些实例，而不是这些抽象产品的实例，换言之，也就是这些抽象产品的具体子类的实例。这就是抽象工厂模式用意的基本含义。
第三段：抽象工厂模式提供两个具体工厂角色，分别对应于这两个具体产品角色。每一个具体工厂角色仅负责某一个具体产品角色的实例化。每一个具体工厂类负责创建抽象产品的某一个具体子类的实例
“抽象工厂”——“抽象”来自“抽象产品角色”，而“抽象工厂”就是抽象产品角色的工厂。

3、抽象工厂模式与工厂方法模式最大的区别在于，工厂方法模式针对的是一个产品等级结构，而抽象工厂模式则需要对多个产品等级结构

4、产品族（Product Family）：是指位于不同产品等级结构中，功能相关联的产品组成的家族。每一个产品族中含有产品的数目，与产品等级结构的数目是相等的

5、抽象工厂模式是对象的创建模式，它是工厂方法模式的进一步推广。
假设一个子系统需要一些产品对象，而这些产品又属于一个以上的产品等级机构。那么为了将消费这些产品对象的责任和创建这些产品对象的责任分割开来，可以引用抽象工厂模式。这样的话，消费产品的一方不需要直接参与产品的创建工作，而只需要向一个公用的工厂接口请求所需要的产品

6、抽象工厂模式涉及到的角色：
1) 抽象工厂角色：担任这个角色的是工厂方法模式的核心，它是与应用系统的商业逻辑无关的。通常使用java接口或者抽象java类实现，而所有的具体工厂类必须实现这个java接口或继承这个这个抽象java类
2）具体工厂类角色; 这个角色直接在客户端的调用下创建产品的实例，这个角色含有选择合适的产品对象的逻辑，而这个逻辑是与应用系统的商业逻辑紧密相关的。通常使用具体java类实现这个角色
3）抽象产品角色：担任这个角色的类是工厂方法模式所创建的对象的父类，或他们共同拥有的接口。通常使

用java接口或者抽象java类实现这一角色

4）具体产品角色：抽象工厂模式所创建的任何产品对象都是某一个具体产品类的实例。这是客户端最终需要的东西，其内部一定充满了应用系统商业逻辑。通常使用java类实现这个角色

```
//抽象产品角色
        public interface Creator{
                产品等级结构A的工厂方法
                public ProductA factoryA();

                public ProductB factoryB();
        }

        //具体工厂类1
        public class ConcreateCreator1 implements Creator{
                public ProductA factoryA(){
                        return new ProductA1();
                }

                public ProductB factoryB(){
                        return new ProductB1();
                }
        }
一般而言，有多少个产品等级结构，就会在工厂角色中发现多少个工厂方法，每一个产品等级结构中有多少具体产品，
        public class ConcreateCreator2 implements Creator{
                public ProductA factoryA(){
                        return new ProductA2();
                }

                public ProductB factoryB(){
                        return new ProductB2();
                }
        }
        //抽象产品类
        public interface ProductA{

        }
```

```
public interface ProductB{

}
//具体产品类
public class ProductA1 implements ProductA{
        public ProductA1(){

        }
}

public class ProductA2 implements ProductA{
        public ProductA2(){

        }
}

public class ProductB1 implements ProductB{
        //构造子
        public ProductB1(){

        }
}

public class ProductB2 implements ProductB{
        public ProductB2(){

        }
}
```

7、在以下情况下应当考虑使用抽象工厂模式：

1）一个系统不应当依赖于产品类实例如何被创建、组合和表达的细节，这对于所有形态的工厂模式都是重要的

2）这个系统的产品有多余一个的产品族，而系统之消费其中某一族的产品

3）同属于同一个产品族的产品是在一起使用的，这一约束必须在系统的设计中体现出来

4）系统提供一个产品类的库，所有的产品以同样的接口出现，从而使客户端不依赖于实现

```
//抽象工厂类
		public interface Gardener{

		}

		//具体工厂类
		public class NorthernGardener implements Gardener{
			//水果的工厂方法
			public Fruit createFruit(String name){
				return new NorthernFruit(name);
			}

			public Veggie createVeggie(String name){
				return new NorthernVeggie(name);
			}
		}

		public class TropicalGardener implements Gardener{
			public Fruit createFruit(String name){
				return new TropicalFruit(name);
			}

			public Veggie createVeggie(String name){
				return new TropicalVeggie(name);
			}
		}

		//抽象产品类
		public interface Veggie{}

		public interface Fruit{}

		//具体产品类
		public class NorhthernVeggie implements Veggie{
			private String name;
			public NorthernVeggie(String name){
				this.name = name;
```

```
        }
        public String getName(){
                return name;
        }
        public void setName(String name){
                this.name = name;
        }
}

public class TropicalVeggie implements Veggie{
        private String name;
        public TropicalVeggie(String name){
                this.name = name;
        }

        public String getName(){
                return name;
        }

        public void SetName(String name){
                this.name = name;
        }
}

public class NorthernFruit implements Fruit{
        private String name;
        public NorthernFruit(String name){
                this.name = name;
        }

        public String getName(){
                return name;
        }

        public void setName(String name){
                this.name = name;
        }
```

```
        }

        public class TropicalFruit implements Fruit{
                private String name;
                public TropicalFruit(String name){
                        this.name = name;
                }

                public String getName(){
                        return name;
                }

                public void setName(String name){
                        this.name = name;
                }
        }
//在使用时，客户端只需要创建具体工厂的实例，然后调用工厂对象的工厂方法，就可以得到所需要的产品对象
```

8、 “开-闭”原则

“开-闭”原则要求一个软件系统可以在不修改原有代码的情况下，通过扩展达到增强其功能的目的。对于一个涉及到多个产品等级结构和多个产品族的系统，其功能的增强不外乎两方面：

1）增加新的产品族

2）增加新的产品等级结构

1、增加新的产品族：在产品等级结构的数目不变的情况下，增加新的产品族，就意味着在每一个产品等级结构中增加一个（或者多个）新的具体（或者抽象和具体）产品角色。所以设计师只需要向系统中加入新的具体工厂类就可以了，没有必要修改已有的角色产品角色。因此，在系统中的产品族增加时。抽象工厂模式是支持“开——闭”原则的。

2、增加新的产品等级结构：在产品族的数目不变的情况下，增加新的产品等级结构。所有的产品等级结构中的产品数目不会改变，但是现在多出一个与现有的产品等级结构平行的新的产品等级结构，要做到这一点，就需要修改所有的工厂角色，给每一个工厂类都增加一个新的工厂方法，而这显然是违背“开-闭”原则的。换言之，对于产品等级结构的增加，抽象工厂模式是不支持“开-闭”原则的。

综合起来，抽象工厂模式以一种倾斜的方式支持增加新的产品，它为新的产品族的增加提供方便，而不能为新

的产品等级结构增加提供这样
的方便。

1.2 java设计模式笔记【创建模式第二篇】

发表时间: 2009-10-23

工厂模式的几种形态：

1、简单工厂模式，又叫做静态工厂方法（Static Factory Method）模式。

2、工厂方法模式，又称为多态性工厂(Polymorphic Factory)模式

3、抽象工厂模式，又称工具（Kit或ToolKit）模式

简单工厂模式（Simple Factory）

1、模式：

简单工厂模式是类的创建模式，又叫做静态工厂方法（Static Factory Method）模式。

它是由一个工厂对象决定创建出哪一种产品类的实例。

2、举例：

```
//水果接口
public interface Fruit{
        void grow();            //生长
        void harvest(); //收获
        void plant();           //种植
}

//苹果类
public class Apple implements Fruit{
        private int treeAge     //树龄

        public void grow(){
                log("Apple is growing...");
        }

        public void harvest(){
                log("Apple has been harvented.");
        }

        public void plant(){
                log("Apple has been planted.");
        }
```

```
        //辅助方法
        public static void log(String msg){
                System.out.println(msg);
        }

        public int getTreeAge(){
                return treeAge;
        }

        public void setTreeAge(int treeAge){
                this.treeAge = treeAge;
        }
}

//葡萄类
public class Grape implements Fruit{
        private boolean seedless;

        public void grow(){
                log("Grape is growing...");
        }

        public void harvest(){
                log("Grape has been harvested.");
        }

        public void plant(){
                log("Grape has been planted.");
        }

        public static void log(String msg){
                System.out.println(msg);
        }

        //有无籽方法
        public boolean getSeedless(){
```

```
                return seedless;
        }

        public void setSeedlees(boolean seedless){
                this.seedless = seedless;
        }
}

//草莓类
public class Strawberry implements Fruit{
        public void grow(){
                log("Strawberry is growing...");
        }

        public void harvest(){
                log("Strawberry has been harvested.");
        }

        public void plant(){
                log("Strawberry has been planted.");
        }

        public static void log(String msg){
                System.out.println(msg);
        }
}

//农场园丁类，由他决定创建哪种水果类的实例
public class FruitGardener{
        //静态工厂方法
        public static Fruit factory(String which) throws BadFruitException{
                if(which.equalslgnoreCase("apple")){
                        return new Apple();
                }else if(which.equalslgnoreCase("grape")){
                        return new Grape();
                }else if(which.equalslgnoreCase("strawberry")){
                        return new Strawberry();
```

```
                }else{
                        throw new BadFruitException("Bad fruit request");
                }
        }
}

//异常类
public class BadFruitException extends Exception{
        public BadFruitException(String msg){
                super(msg);
        }
}

//测试类
public class Test{
        FruitGardener gardener =new FruitGardener();
        try{
                gardener.factory("grape");
                gardener.factory("apple");
                gardener.factory("strawberry");
                gardener.factory("banana");     //抛出异常
        }catch(BadFruitException e){
                System.out.println("dont't has this fruit.");
        }
}
```

3、三个角色：

1、工厂类（Creator）角色：在客户端的直接调用下创建产品对象

2、抽象产品角色：可以用一个接口或者一个抽象类实现

3、具体产品角色：工厂方法模式所创建的任何对象都是这个角色的实例

4、如果工厂方法总是循环使用同一个产品对象，那么这个工厂对象可以使用一个属性来存储这个产品对象。

每一次客户端调用工厂方法时，工厂方法总是提供这同一个对象。

如果工厂方法永远循环使用固定数目的一些产品对象，而且这些产品对象的数目并不大的话，

可以使用一些私有属性存储对这些产品对象的引用。比如：一个永远只提供一个产品对象的工厂对象

可以使用一个静态变量存储对这个产品对象的引用。

如果工厂方法使用数目不确定，或者数目较大的一些产品对象的话，使用属性变量存储对这些产品对象的引用就不方便了，

这时候 就应当使用聚集对象存储对产品对象的引用。

5、其他模式;

1、单例模式：单例模式要求单例类的构造子是私有的，从而客户端不能直接将之实例化，而必须通过这个静态工厂方法将之实例化，

而且单例类自身是自己的工厂角色。单例类自己负责创建自身的实例

单例类使用一个静态属性存储自己的唯一的实例，工厂方法永远仅提供这一个实例

2、多例模式：它是对单例模式的推广，多例模式也禁止外界直接将之实例化，同时通过静态工厂方法想外界提供循环使用的自身的实例，

多例模式可以有多个实例

多例模式具有一个聚集属性，通过向这个聚集属性登记已经创建过的实例达到循环使用实例的目的，

它还拥有一个内部状态，每一个内部状态都只有一个实例存在

根据外界传入的参量，工厂方法可以查询自己的登记聚集，如果具有这个状态的实例已经存在，

就直接将这个实例提供给外界：反之，就首先创建一个新的满足要求的实例，将之登记到聚集中，然后再提供客户端。

3、备忘录模式：单例模式和多例模式使用一个属性或者聚集属性来登记所创建的产品对象，一边可以通过查询这个属性或者聚集属性

找到并共享已经创建了的产品对象，这就是备忘录模式的应用。

4、MVC模式：是更高层次上的架构模式。

包括：合成模式、策略模式、观察者模式、也有可能会包括装饰模式、调停者模式、迭代子模式以及工厂方法模式等。

* * * * ：如果系统需要有多个控制器参与这个过程的话，简单工厂模式就不适合了，应当考虑使用工厂方法模式。

6、简单工厂模式的优点己缺点

优点：模式的核心是工厂类，这个类有必要的判断逻辑，可以决定在什么时候创建哪一个产品类的实例。

而客户端则可以免除直接创建产品对象的责任，而仅仅负责“消费”产品。简单工厂模式通过这种做法实现了对责任的分割

缺点：1)这个工厂类集中了所有的产品创建逻辑，形成一个无所不知的全能类，有人把这种类叫做上帝类（God Class）。

如果这个全能类代表的是农场的一个具体园丁的话，那么这个园丁就需要对所有的产品负责，成了农场的关键人物，

他什么时候不能正常工作了，整个农场都要受到影响；

2)将这么多的逻辑集中放在一个类里面的另外一个缺点是，当产品类有不同的接口种类时，工厂需要判断在什么时候创建某种产品，

这种对时机的判断和对哪一种具体产品的判断逻辑混合在一起，使得系统在将来进行功能扩展时较为困难。

这一缺点在工厂方法模式中得到克服

3）由于简单工厂模式使用静态方法作为工厂反尬，而静态方法无法由子类继承，因此工厂角色无法形成基于继承的等级结构。

这一缺点会在工厂方法模式中得到克服

7、“开——闭”原则要求一个系统的设计能够允许系统在无需修改的情况下，扩展其功能。

要求系统允许当新的产品加入系统中时，而无需对现有代码进行修改，这一点对于产品消费者角色是成立的，而对于工厂角色是不成立的

一般而言：一个系统总是可以划分成为产品的消费者角色（Client）、产品的工厂角色（Factory）以及产品角色（Product）三个子系统

对于产品消费者角色来说，任何时候需要某种产品，只需向工厂角色请求即可，而工厂角色在接到请求后，

会自行判断创建和提供哪一个产品，所以，产品消费者角色无需知道它得到的是哪一个产品，

产品消费者角色无需修改就可以接纳新的产品，而接纳新的产品意味着要修改这个工厂角色的源代码

（简单工厂角色只在有限的程度上支持“开—闭”原则）

1.3 java设计模式笔记【创建模式第三篇】

发表时间: 2009-10-23

工厂方法（Factory Method）模式

1、工厂方法模式是类的创建模式，又叫做虚拟构造子模式或者多态性工厂模式

工厂方法模式的用意是定义一个创建产品对象的工厂接口，将实际创建工作推迟到子类中。

一般而言工厂方法模式的系统设计到以下几种角色：

1）抽象工厂（Creator）角色：担任这个角色的是工厂方法模式的核心，它是与应用程序无关的，

任何在模式之中个窗对象的工厂类必须实现这个接口，在实际系统中，这个角色也使用抽象类实现

2）具体工厂（Concrete Creator）角色：担任这个角色的是实现了抽象工厂接口的具体java类。

具体工厂角色含有与用应密切相关的逻辑，并且受到应用程序的调用以创建产品对象。

3）抽象产品角色：工厂方法模式所创建的对象的超类型，也就是产品对象的共同父类或共同拥有的接口，

在实际应用中，这个角色也常常使用抽象java类实现

4）具体产品角色：这个角色实现了抽象产品角色所声明的接口。工厂方法模式所创建的每一个对象都是某个具体产品角色的实例

2、举例：

```
        //抽象工厂（角色）接口
        public interface Creator{
                //工厂方法
                public Product factory();
        }

        //抽象产品（角色）接口
        public interface Product{

        }                                   //一个没有声明任何方法的接口叫做标识接口

        //具体工厂（角色）类
        public class ConcreteCreator1 implements Creator{
                //工厂方法
                public Product factory(){
                        return new ConcreteProduct1();
```

```
        }
}

public class ConcreteCreator2 implements Creator{
        //工厂方法
        public Product factory(){
                return new ConcreteProduct2();
        }
}

//具体产品(角色)类
public class ConcreteProduct1 implements Product{
        public ConcreteProduct1(){
                //do somthing
        }
}

public class ConcreteProduct2 implements Product{
        public ConcreteProduct2(){
                //do somthing
        }
}

//客户端(角色)类
public class Client{
        private static Creator creator1,creator2;
        private static Product prod1,prod2;

        public static void main(String args[]){
                creator1 = new ConcreteCreator1();
                prod1 = creator1.factory();
                creator2 = new ConcreteCreator2();
                prod2 = creator2.factory();
        }
}
```

Client对象的活动可以分为两部分

1）客户端创建 ConcreteCreator1 对象，这时客户端所持有变量的静态类型是Creator，而实际类型是ConcreteCreator1，然后，客户端调用 ConcreteCreator1 对象的工厂方法 factory() ，接着后者调用 ConcreteProduct1 的构造子创建出产品对象

2、举例：

```
//抽象工厂角色（水果园丁）
public interface FruitGardener{
        //工厂方法
        public Fruit factory();
}

//具体工厂类（苹果园丁）
public class AppleGardener implements FruitGardener{
        public Fruit factory(){
                return new Apple();
        }
}

//.....（草莓园丁）
public class StrawberryGardener implements FuritGardener{
        public Fruit factory(){
                return new Strawberry();
        }
}

//.....（葡萄园丁）
public class GrapeGardener implements FuritGardener{
        public Fruit factory(){
                return new Grape();
        }
}

//抽象产品角色（水果接口）
public interface Fruit{
        void grow();
```

```
        void harvest();
        void plant();
}

//具体产品角色（苹果类）
public class Apple implements Fruit{
        private int treeAge;

        public void grow(){
                log("Apple is growing...");
        }

        public void harvest(){
                log("Apple has been harvested.");
        }

        public void plant(){
                log("Apple has been planted.");
        }

        public static void log(String msg){
                System.out.println(msg);
        }

        public int getTreeAge(){
                return treeAge;
        }

        public void setTreeAge(int treeAge){
                this.treeAge = treeAge;
        }
}
//其他同上.....................
```

3、工厂方法返还的类型

工厂返还的应当是抽象类型，而不是具体类型，只有这样才能保证整队产品的多态性，调用该工厂方法的客户

端可以针对抽象编程，依赖于一个抽象产品类型，而不是具体产品类型。
在特殊情况下，工厂方法仅仅返还一个具体产品类型，这个时候工厂方法模式的功能就退化了，表现为针对产品角色的多态性的丧失，客户端从工厂方法的静态类型可以知道将要得到的是什么类型的对象，而这违背了工厂方法模式的用意

4、与其他模式的关系

1、模板方法模式
工厂方法模式常常与模板方法模式一起联合使用，其原因有二：
一、两个模式都是基于方法的，工厂方法模式是基于多态性的工厂方法的，而模板方法模式是基于模板方法和基于方法的
二、两个模式的哦将具体工作交给子类，工厂方法模式将创建工作推延给子类，模板方法模式将生于逻辑交给子类

2、MVC模式
工厂方法模式总是设计到两个等级结构中的对象，而这两个等级结构可以分别是MVC模式中的控制器和视图。一个MVC模式可以有多个控制器和多个视图，控制器端可以创建合适的视图断，如同工厂角色创建合适的对象角色一样，模型端则可以充当这个创建过程的客户端

3、亨元模式
使用了带有逻辑的工厂方法

4、备忘录模式
亨元模式使用了一个聚集来等级所创建的产品对象，以便可以通过查询这个聚集找到和共享已经创建了的产品对象，这就是备忘录模式的应用

5、举例：

问题：某一个商业软件产品需要支持Sybase和Oracle数据库。这个系统需要这样一个查询运行期系
答案：可以看出，这个系统是由一个客户端Client，一个抽象工厂角色QueryRunner，两个具体工
对于客户端Client而言，系统的抽象产品角色是ResultSet接口，而具体产品角色就是java.sql.

```
//java抽象类QueryRunner
import java.sql.Connection;
import java.sql.ResultSet;

public abstract class QueryRunner{
        public ResultSet run() throws Exception{
                Connection conn = createConnection();
                String sql = createSql();
```

```
                return runSql(conn,sql);
        }
        protected abstract Connection createConnection();
        protected abstract String createSql();
        protected abstract ResultSet runSql(Connection conn.String sql)throws E
}

//具体工厂类
import java.sql.Connection;
import java.sql.Statement;
import java.sql.ResultSet;

public class SybaseQueryRunner extends QueryRunner{
        public Connection createConnection(){
                return null;
        }

        protected String createSql(){
                return "select * from customers";
        }

        protected ResultSet runSql(Connection conn,String sql) throws Exceptior
                Statement stmt = conn.createStatement();
                return stmt.executeQuery(sql);
        }
}

//具体工厂类
import java.sql.Connection;
import java.sql.Statement;
import java.sql.ResultSet;

public class OracleQueryRunner extends QueryRunner{
        public Connection createConnection(){
                return null;
        }
```

```
        protected String createSql(){
                return "select * from customers";
        }

        protected ResultSet runSql(Connection conn, String sql) throws Exceptic
                Statement stmt = conn.createStatement();
                return stmt.executeQuery(sql);
        }
}

//客户端
import java.sql.ResultSet;

public class Client{
        private static QueryRunner runner;

        public static void main(String args[]) throws Exception{
                runner = new SybaseQueryRunner();
                ResultSet rs = runner.run();
        }
}
```

以上给出的答案中，使用了模板方法模式，在QueryRunner中，run()方法就是一个模板方法，这个

1.4 java设计模式笔记【创建模式第四篇】

发表时间: 2009-10-23

单例（Singleton）模式:单例模式确保某一个类只有一个实例，而且自行实例化并向整个系统提供这个实例。这个类称为单例类

1、单例模式的要点：

1）某个类只能有一个实例

2）它必须自行创建这个实例

3）它必须自行向整个系统提供这个实例

2、单例模式的例子：

1）打印机

2）传真机使用的传真卡，只应该由一个软件管理

3）Windows回收站

3、单例模式有以下特点：

1）单例类只能有一个实例

2）单例类必须自己创建自己的唯一的实例

3）单例类必须给所有其他对象提供这一实例

```
* 饿汉式单例类：
                        public class EagerSingleton{
                                private static final EagerSingleton m_instance = new EagerSingl

                                private EagerSingleton(){}

                                public static EagerSingleton getInstance(){
                                        return m_instance;
                                }
                        }
                java语言中单例类的一个最重要的特点是类的构造子是私有的，从而避免外界利用构造子直接创建
                因为构造子是私有的，因此此类不能被继承

                * 懒汉式单例类：
                        与饿汉式单例类相同的是，类的构造子是私有的，不同的是在第一次被引用时将自己实例
                        式单例类被加载时不会将自己实例化。
                        public class LazySingleton{
                                private static LazySingleton m_instance = null;
```

```
                private LazySingleton(){}

                synchronized public static LazySingleton getInstance(){
                        if(m_instance == null){
                                m_instance = new LazySingleton();
                        }
                        return m_instance;
                }
        }
```

* 登记式单例类

```
        public class RegSingleton{
                private static HashMap m_registry = new HashMap();

                static{
                        RegSingleton x = new RegSingleton();
                        m_registry.put(x.getClass().getName(),x);
                }

                protected RegSingleton(){}

                public static RegSingleton getInstance(String name){
                        if(name == null){
                                name = "RegSingleton";
                        }
                        if(m_registry.get(name) == null){
                                try{
                                        m_registry.put(name,Class.forName(name)
                                }catch(Exception e){
                                        System.out.println("Error happened.");
                                }
                        }
                        return (RegSingleton)(m_registry.get(name));
                }
                //一个示意性的商业方法
                public String about(){
```

```
                return "Hello, I am RegSingleton";
        }
    }

    //登记式单例类的子类
    public class RegSingletonChild extends RegSingleton{
        public RegSingletonChild(){}

        //静态工厂方法
        public static RegSingletonChild getInstance(){
            return (RegSingletonChild)RegSingleton.getInstance("Reg
        }

        public String about(){
            return "Hello, I am RegSingletonChild";
        }
    }
```

4、使用单例模式的条件：

使用单例模式有一个必要条件：在一个系统要求一个类只有一个实例时才应当使用单例模式，反过
存，那么就没有必要使用单例模式类。

```
//属性管理器
import java.util.Properties;
import java.io.FileInputStream;
import java.io.File;

public class ConfigManager{
    //属性文件全名
    private static final String PFILE = System.getProperty("user.dir")
    + File.Separator + "Singleton.properties";
    //对应于属性文件的文件对象变量
    private File m_file = null;
```

```
//属性文件的最后修改日期
private long m_lastModifiedTime = 0;
//属性文件所对应的属性对象变量
private Properties m_props = null;
//本类可能存在的唯一的一个实例
private static ConfigManager m_instance = new ConfigManager();
//私有的构造子，用以保证外界无法直接实例化
private ConfigManager(){
        m_file = new File(PFILE);
        m_lastModifiedTime = m_file.lastModified();
        if(m_lastModifiedTime == 0){
                System.out.println(PFILE + "file does not exits!");
        }
        m_props = new Properties();
        try{
                m_props.load(new FileInputStream(PFILE));
        }catch(Exception e){
                e.printStackTrace();
        }
}
//静态工厂方法
public synchronized static ConfigManager getInstance(){
        return m_instance;
}
//读取一个属性项
//@name_名称，defaultVal_属性项的默认值，
public final Object getConfigItem(String name, Object defaultVal){
        long newTime = m_file.lastModified();
        if(newTime == 0){
                if(m_lastModifiedTime == 0){
                        System.err.println(PFILE + "file does not exits
                }else{
                        System.err.println(PFILE + "file was deleted!")
                }
                return defaultVal;
        }else if(newTime > m_lastModifiedTime){
                m_props.clear();
```

```
                                try{
                                        m_props.load(new FileInputStream(PFILE));
                                }catch(Exception e){
                                        e.printStackTrace();
                                }
                        }
                        m_lastModifiedTime = newTime;
                        Object val = m_props.getProperty(name);
                        if(cal == null){
                                return defaultVal;
                        }else{
                                return val;
                        }
                }
        }
```

5、不完全单例类

```
        public class LazySingleton{
                private static LazySingleton m_instance = null;
                //公开的构造子
                public LazySingleton(){}

                public static synchronized LazySingleton getInstance(){
                        if(m_instance == null){
                                m_instance = new LazySingleton();
                        }
                        return m_instance;
                }
        }
```

6、相关模式：

1）简单工厂模式：单例模式使用了简单工厂模式来提供自己的实例

最常见的目录服务包括LDAP和DNS————（命名—服务）将一个对象与一个名字联系起来，使得客户可以通过对象的名字找到这个对象

目录服务允许对象有属性，这样客户端通过查询找到对象后，可以进一步得到对象的属性，或者反过来根据属

性查询对象

DNS————即域名服务，电脑用户在网络上找到其他的电脑用户必须通过域名服务。在国际网络以及任何一个建立在TCP / IP基础上的
网络上的，每一台电脑都有一个唯一的IP地址。

MX记录——就是邮件交换记录。电子邮件服务器记录指定一台服务器接受某个域名服务器的邮件，也就是说，发往jeffcorp.com 的邮
件将发往 mail.jeffcorp.com 的服务器，完成此任务的MX纪录应当像下面这样：
jeffcorp.com.IN MX 10 mail.jeffcorp.com
在上面的这个记录的最左边是国际网络上使用的电子邮件域名，第三列是一个数字10，它代表此服务器的优先权是10。通常来说，一个
大型的系统会有数台电子邮件服务器，这些服务器可以依照优先权作为候补服务器使用，优先权不需是一个正整数，这个数字越低，表明
优先权越高。

JNDI的全称是java命名和地址界面，其目的是为java系统提供支持各种目录类型的一个一般性的访问界面

JNDI架构由JNDI API 和 JNDI SPI组成。JNDI API使得一个java应用程序可以使用一系列的命名和目录服务。JNDI SPI是为服务提
供商，包括目录服务提供商准备的，它可以让各种命名和目录服务能够以对应用程序透明的方式插入到系统里。

JNDI API由以下四个库组成：
1）javax.naming: 包括了使用命名服务的类和接口。
2）javax.naming.directory: 扩展了核心库javax.naming的功能，提供目录访问功能。
3）javax.naming.event: 包括了命名和目录服务中所需要的时间同志机制的类和接口
4）javax.naming.ldap: 包括了命名和目录服务中支持LDAP（v3）所需要的类和接口
JNDI SPI的库为：
javax.naming.spi: 包括的类和接口允许各种的命名和目录服务提供商能将自己的软件服务构建加入到JNDI架构中去
在JNDI里，所有的命名和目录操作都是相对于某一个context(环境)而言，JNDI并没有任何的绝对根目录。
JNDI定义一个初始环境对象
称为InitialContext,来提供命名和目录操作的起始点。一旦得到了初始环境，就可以使用初始环境查询其他环境和对象。

```
//MailServer
                public class MailServer{
```

```
        private int priority;
        private String server;
        //优先权的赋值方法
        public void setPriority(int priority){
                this.priority = priority;
        }
        //服务器名的赋值方法
        public void setServer(String server){
                this.server = server;
        }
        public int getPriority(){
                return priority;
        }
        public String getServer(){
                return server;
        }
}

//系统核心类（MXList）
import java.util.Enumeration;
import java.util.Hashtable;
import java.util.Vector;
import java.util.String.Tokenizer;
import javax.naming.NamingEnumeration;
import javax.naming.directory.DirContext;
import javax.naming.directory.InitialDirContext;
import javax.naming.directory.Attributes;
import javax.naming.directory.Attribute;

public class MXList{
        private static MXList mxl = null;
        private Vector list = null;
        private static final String FACTORY_ENTRY = "java.naming.factory.initia
        private static final String FACTORY_CLASS = "com.sun.jndi.dns.DnsContex
        private static final String PROVIDER_ENTRY = "java.naming,provider.url'
        private static final String MX_TYPE = "MX";
        private String dnsUrl = null;
```

```
        private String domainName = null;

        private MXList(){}

        private MXList(String dnsUrl, String domainName) throws Exception{
                this.dnsUrl = dnsUrl;
                this.domainName = domainName;

                this.list = getMXRecords(dnsUrk, domainName);
        }

        //静态工厂方法
        public static synchronized MXList getInstance(String dnsUrl,String doma
                if(mxl == null){
                        mxl = new MXList(dnsUrl, domainName);
                }
                return mxl;
        }
        //聚集方法，提供聚集元素
        public MailServer elementAt(int index) throws Exception{
                return (MailServer)list.elementAt(index);
        }
        //聚集方法，提供聚集大小
        public int size(){
                return list.size();
        }
        //辅助方法，向DNS服务器查询MX记录
        private Vector getMXRecords(String providerUrl, String domainName) thro
                //设置环境性质
                Hashtable env= new Hashtable();
                env.put(FACTORY_ENTRY, FACTORY_CLASS);
                env.put(PROVIDER_ENTRY, providerUrl);

                //创建环境对象
                DirContext dirContext = new InitialDirContext(env);

                Vector records = new Vector(10);
```

```
                        //读取环境对象的属性
                        Attributes attrs = dirContext.getAttributes(domainName,new Stri

                        for(NamingEnumeration ne = attrs.getAll(); ne != null && ne.has
                                Attribute attr = (Attribute)ne.next();
                                NamingEnumeration e = attr.getAll();
                                while(e.hasMoreElements()){
                                        String element = e.nextElement().toString();
                                        StringTokenizer tokenizer = new StringTokenizer

                                        MailServer mailServer = new MailServer();

                                        String token1 = tokenizer.nextToken();
                                        String token2 = tokenizer.nextToken();

                                        if(token1 != null && token2 != null){
                                                mailServer.setPriority(Integer.valueOf(
                                                mailServer.setServer(token2);
                                                records.addElement(mailServer);
                                        }
                                }
                        }
                        System.out.println("List created.");
                        return records;
                }

        }

        //客户端类
        public class Client{
                private static MXList mxl;
                public static void main(String args[]) throws Exception{
                        mxl = MXList.getInstance("dns://dns01390.ny.jeffcorp.com", "jef

                        for(int i = 0; i < mxl.size(); i ++){
                                System.out.println((i + 1) + ") priority = " + ((MailSe
```

```
                                    getPriority() + ", Name = " + ((MailServer)mxl.elementA
                            }
                    }
            }
```

1.5 java设计模式笔记【创建模式第五篇】

发表时间: 2009-10-23

多例模式（Multiton Pattern）

1、多例模式特点：

1）多例类可有多个实例

2）多例类必须自己创建、管理自己的实例，并向外界提供自己的实例。

```
//多例类（骰子）
                import java.util.Random;
                import java.util.Date;

                public class Die{
                        private static Die die1 = new Die();
                        private static Die die2 = new Die();

                        private Die(){
                        }

                        //工厂方法
                        public static Die getInstance(int whichOne){
                                if(whichOne == 1){
                                        return die1;
                                }else{
                                        return die2;
                                }
                        }

                        //掷骰子，返回一个在1～6之间的随机数
                        public synchronized int dice(){
                                Date d = new Date();
                                Random r = new Random(d.getTime());
                                int value = r.nextInt();
                                value = Math.abs(value);
                                value = value % 6;
                                value += 1;
                                return value;
```

```
        }
}

//客户端
public class Client{
        private static Die die1, die2;
        public static void main(String args[]){
                die1 = Die.getInstance(1);
                die2 = Die.getInstance(2);
                die1.dice();
                die2.dice();
        }
}

//多例类是单例类的推广，而单例类是多例类的特殊情况
多例类对象的状态如果是可以在加载后改变的，那么这种多例对象叫做可边多例对象；如果多例对
那么这种多例对象叫做不变多例对象

//怎样使用Locale对象和ResourceBundle对象
Locale locale = new Locale("fr", "FR");
ResourceBundle res = ResourceBundle("shortname",locale);
res会加载一个名为shortname_fr_FR.properties的Resource文件

//多例类LingualResource
import java.util.HashMap;
import java.util.Locale;
import java.util.ResourceBundle;

public class LingualResource{
        private String language = "en";
        private region = "US";  //region（区域）
        private String localeCode = "en_US";
        private static HashMap instances = new HashMap(19);
        private Locale locale = null;
        private ResourceBundle resourceBundle = null;
        private LingualResource lnkLingualResource;
```

```
        private LingualResource(String language, String region){
                this.localeCode = language;
                this.region = region;
                localeCode = makeLocaleCode(language, region);
                locale = new Locale(language, region);
                resourceBundle = ResourceBundle.getBundle(FILE_NAME, locale);
                instances.put(makeLocaleCode(language, region), resourceBundle)
        }

        private LingualResource(){
        }

        //工厂方法，返回一个具体的内部状态的实例
        public synchronized static LingualResource getInstance(String language,
                if(makeLocaleCode(language, region)){
                        return (LingualResource)makeLocaleCode(language, regior
                }else{
                        return new LingualResource(language, region);
                }
        }

        public String getLocaleString(String code){
                return resourceBundle.getString(code);
        }

        private static String makeLocaleCode(String language, String region){
                return language + "_" + region;
        }
}

//makeLocaleCode()是一个辅助性的方法，在传入语言代码和地区代码时，此方法可以返回一个L
在getInstance()方法被调用时，程序会首先检查传入的Locale代码是否已经在instances集合中
即直接返回同它所对应的LingualResource对象，否则就会首先创建一个此Locale代码所对应的L
instances集合，并返回这个实例

//客户端
public class LingualResourceTester{
```

```
        public static void main(String agrs[]){
                LingualResource ling = LingualResource.getInstance("en", "US");
                String usDollar = ling.getLocaleString("USD");
                System.out.println("USD = " + usDollar);
                LingualResource lingZh = LingualResource.getInstance("zh", "CH"
                String usDollarZh = lingZh.getLocaleString("USD");
                System.out.println("USD = " + usDollarZh);
        }
}
//如果客户是美国客户，那么在jsp网页中可以通过调用getLocalString()方法得到相应的英文说
Resource文件res_en_US.properties的内容
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
                LingualResource ling = LingualResource.getInstance("en", "US");
                String usDollar = ling.getLocaleString("USD");
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
返回 - - - - US Dollar

中国用户：
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
                LingualResource lingZh = LingualResource.getInstance("zh", "CH"
                String usDollarZh = lingZh.getLocaleString("USD");
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
返回 - - - - 美元

/ / / / / / / / / / / / / / / / / / / / / / / / / / / / / / / / / / / / / / / /
为美国英文准备的Resource文件res_en_US.properties的内容如下：
        USD = US Dollar
        JPY = Japanese Yen
为简体中文准备的Resource文件res_zh_CH.properties的内容如下：
        USD = 美元
        JPY = 日元

= = = = = = = = = = = = = = = = = = = = = = =问答题= = = = = = = = = = = = = = =
一个根据语言代码和地区代码将数字格式化的例子
import java.util.Locale;
import java.text.NumberFormat;
```

```
public class NumberFormatTester{
        public static void displayNumber(Double amount, Locale currentLocale){
                NumberFormat formatter;
                String amountOut;
                formatter = NumberFormat.getNumberInstance(currentLocale);
                amountOut = formatter.format(amount);
                System.out.println(amountOut + "  " + currentLocale.toString())
        }

        public static void main(String agrs[]){
                displayNumber(new Double(1234567.89), new Locale("en", "US"));
                displayNumber(new Double(1234567.89), new Locale("de", "DE"));
                displayNumber(new Double(1234567.89), new Locale("fr", "FR"));
        }
}
```

一个根据语言代码和地区代码将货币数字格式化的例子

```
import java.util.Locale;
import java.text.NumberFormat;

public class CurrencyFormatTester{
        public static void displayCurrency(Double amount, Locale currentLocale)
                NumberFormat formatter;
                String amountOut;
                formatter = NumberFormat.getCurrencyInstance(currentLocale);
                amountOut = formatter.format(amount);
                System.out.println(amountOut + "  " + currentLocale.toString())
        }

        public static void main(String agrs[]){
                displayCurrency(new Double(1234567.89), new Locale("en", "US"))
                displayCurrency(new Double(1234567.89), new Locale("de", "DE"))
                displayCurrency(new Double(1234567.89), new Locale("fr", "FR"))
        }
}
```

一个根据语言代码和地区代码将百分比格式化的例子

```
import java.util.Locale;
import java.text.NumberFormat;

public class PercentFormatTester{
        public static void displayPercent(Double amount, Locale currentLocale){
                NumberFormat formatter;
                String amountOut;
                formatter = NumberFormat.getPercentInstance(currentLocale);
                amountOut = formatter.format(amount);
                System.out.println(amountOut + "  " + currentLocale.toString())
        }

        public static void main(String agrs[]){
                displayPercent(new Double(4567.89), new Locale("en", "US"));
                displayPercent(new Double(4567.89), new Locale("de", "DE"));
                displayPercent(new Double(4567.89), new Locale("fr", "FR"));
        }
}
```

1.6 java设计模式笔记【创建模式第六篇】

发表时间: 2009-10-23

建造（Builder）模式
建造模式是对象的创建模式。
产品的内部表象：一个产品常有不同的组成成分作为产品的零件，这些零件有可能是对象，也有可能不是对象它们通常叫做产品的内部表象

角色：
1、抽象建造者角色：给出一个抽象接口，以规范产品对象的各个组成成分的建造。一般而言，有多少个零件就有多少个建造方法

2、具体建造者角色：担任这个角色的是与应用程序紧密相关的一些类，它们在应用程序的调用下创建产品的实例。
这个角色完成的人物包括：
1）实现抽象建造者Builder所声明的接口，给出一步一步地完成创建产品实例的操作。
2）在建造过程完成后，提供产品的实例

3、导演者角色：担任这个角色的类调用具体建造者角色以创建产品对象。导演者角色并没有产品类的具体知识，真正拥有产品类
具体知识的是具体建造者角色。

4、产品角色：产品便是建造中的复杂对象。一般来说，一个系统会有多余一个的产品类，而且这些产品类并不一定有共同的接口
而完全可以是不想关联的。
（导演者角色是与客户端打交道的角色）
一般类说：没有一个产品类，就有一个相应的具体建造者类。

```
public class Driector{
                        private Builder builder;

                        //产品构造方法，负责调用各个零件建造方法
                        public void construct(){
                                builder = new ConcreteBuilder();
                                builder.buildPart1();
                                builder.buildPart2();
                                builder.retrieveResult();               //retrieve检索
```

```
        }
}

//抽象建造者
public abstract class Builder{
        //产品零件建造方法
        public abstract void buildPart1();

        public abstract void buildPart2();

        //产品返回方法
        public abstract Product retrieveResult();
}

//具体建造者
public class ConcreteBuilder extends Builder{
        private Product product = new Product();

        //产品返回方法
        public Product retrieveResult(){
                return product;
        }

        //产品零件建造方法
        public void buildPart1(){
                //build the first part of the product
        }

        public void buildPart2(){
                //build the second part of the product
        }
}

//产品类
public class Product{
        //...................
}
```

```
        //如果省略了导演者角色
        //客户端代码
        public class Client{
                private static Builder builder;

                public static void main(String args[]){
                        //创建建造者对象
                        builder = new Builder();

                        //拥有建造者对象的产品构造方法
                        builder.construct();

                        //调用建造者对象的产品返还方法以取得产品
                        Product product = builder.retrieveResult();
                }
        }
```

模板方法模式：

准备一个抽象类，将部分逻辑以具体方法以及具体构造子的形式实现，然后声明一些抽象方法迫使
以不同的方式实现这些抽象方法，从而对剩余的逻辑有不同的实现，这就是模板方法模式

如果系统的要求发生变化，要求有不同的零件生成逻辑时，那么设计师有两种选择：

1、一是修改这个退化的建造模式，将它改回成为完全的建造模式，这当然就要涉及到代
2、二是不修改代码，而是将Builder类扩展到不同的子类，在这些子类里面置换掉需要

如果一个产品对象有着固定的几个零件，而且永远只有这几个零件。此时将产品类于建造类合并

* 在很多情况下，建造模式实际上是将一个对象的性质建造过程外部化到独立的建造者对象中，并通过一个
过程进行协调。

一个简单的发送电子邮件的客户端

```
import java.util.*;
import java.io.*;
import javax.mail.*;
import javax.mail.internet.*;
import javax.activation.*;
```

```
public class MailSender{
        private static MineMessage message;

        public static void main(String args[]){
                //你的SMTP服务器地址
                String smtpHost = "smtp.mycompany.com";
                //发送者的地址
                String from = "joff.yan@mycompany.com";
                //收信者地址
                String to = "ni.hao@youcompany.com";

                Properties props = new Properties();
                props.put("mail.smtp.host", smtpHost);

                Session session = Session.getDefaultInstance(props,null);

                try{
                        InternetAddress[] address = new InternetAddress(to);
                        //创建Message对象
                        message = new MineMessage(session);
                        //建造发件人位元址零件
                        message.setFrom(new InternetAddress(from));
                        //建造收件人位元址零件
                        message.setRecipients(Message.RecipientType.TO,address);
                        //建造主题零件
                        messgae.setSubject("Hello from Jeff");
                        //建造发送时间零件
                        message.setSentDate(new Date());
                        //建造内部零件
                        message.setText("Hello,\nHow are things going?");

                        //发送邮件，相当于产品返回方法
                        Transport.send(message);
                        System.out.println("email has been sent.");
                }catch(Exception e){
                        System.out.println(e);
                }
```

```
        }
}
//////////////////////////////////////////////////////////////////////////
//导演类
public class Director{
        Builder builder;

        public Director(Builder builder){
                this.builder = builder;
        }

        //产品构造方法，负责调用各零件建造方法
        public void construct(String toAddress, String fromAddress){
                this.builder.buildSubject();
                this.builder.buildBody();
                this.builder.buildTo(toAddress);
                this.builder.buildFrom(fromAddress);
                this.builder.buildSendDate();
                this.builder.sendMessage();
        }
}

//抽象建造者类
import java.util.Date;

public abstract class Builder{
        protected AutoMessage msg;

        public Builder(){}

        //主题零件的建造方法
        public abstract void buildSubject();
        //内容零件的建造方法
        public abstract void buildBody();
        //发件人零件的建造方法
        public void buildFrom(String from){
                msg.setFrom(from);
```

```
        }
        //收件人零件的建造方法
        public void buildTo(String to){
                System.out.println(to);
                msg.setTo(to);
        }
        //发送时间零件的建造方法
        public void buildSendDate(){
                msg.setSendDate(new Date());
        }
        //邮件产品完成后，用此方法发送邮件
        //此方法相当于产品返回方法
        public void sendMessage(){
                msg.send();
        }
}

//具体建造类
public class WelcomeBuilder extends Builder{
        private static final String subject = "Welcome to philharmony news group";

        public WelcomeBuilder(){
                msg = new WelcomeMessage();
        }

        //主题零件的建造方法
        public void buildSubject(){
                msg.setSubject(subject);
        }

        public void buildBody(){
                String body = "Congratulations for making the choice!";
                msg.setBody(body);
        }

        //邮件产品建造完成后，发送邮件
        //此方法相当于产品返还方法
```

```
                public void sendMessage(){
                        msg.send();
                }
        }
```

//在以下情况下应当使用建造模式

1）需要生成的产品对象有复杂的内部结构。每一个内部成分本身可以是对象，也可以仅仅是一个对象（即产品对象）的一个组成成分

2）需要生成的产品对象的属性相互依赖。301P

3）在对象创建过程中会使用到系统中的其他一些对象，这些对象在产品对象的创建过程中不易得到

使用建造模式的效果：

1、建造模式的使用使得产品的内部表象可以独立地变化。使用建造模式可以使客户端不必知道产品内部组成的细节

2、每一个Builder都相对独立，而与其他的Builder无关

3、模式所建造的最终产品更易于控制。

建造模式与其他模式的关系

建造模式与抽象工厂模式的区别

1、在抽象工厂模式中，每一次工厂对象被调用时都会返回一个完整的产品对象，而客户端有可能会决定把这些产品组成一个更大个复杂的产品，也有可能不会。建造类则不同，它一点一点地建造出一个复杂的产品，而这个产品

的组装过程就发生在建造者角色内部。建造者模式的客户端拿到的是一个完整的最后的产品

2、换言之，虽然抽象工厂模式与建造模式都是设计模式，但是抽象工厂模式处在更加具体的尺度上，而建造模式则处于更加宏观的尺度上。一个系统可以由一个建造模式和一个抽象工厂模式的工厂角色。工厂模式返回不同产品族的零件，而建造者模式则把它们组装起来。

建造模式与策略模式的区别

1、建造模式在结构上很接近于策略模式，事实上建造模式是策略模式的一种特殊情况，这两种模式的区别在于它们的用意

不同。建造模式适用于为客户端一点一点地建造新的对象，而不同类型的具体建造者角色虽然都拥有相同的接口，但是

它们所创建出来的对象则可能完全不同。策略模式的目的是为算法提供抽象的接口。换言之，一个具体策略类把一个算

法包装到一个对象里面，而不同的具体策略对象为一种一般性的服务提供不同的实现。

建造模式与合成模式的关系

1、产品的对象可以是对象，也可以不是对象，而是对象的某种组成成分。当产品的零件确实是对象时，产品对象就变成了

复合对象，因为产品内部还含有子对象。这种对象内含有子对象的结构，可以使用合成模式描述。

2、换言之，合成模式描述了一个对象树的组成结构，而建造模式则可以用来描述对象树的生成过程。

“一二三四五六七八九十”这是古代女子写给心上人的情诗，起含义：

“一别之后，两地相思，三月桃花随水转，四月琵琶未黄，奴我欲对镜心却乱。”

“五月石榴红似火，偏遇冷雨浇花端。六月伏天人热摇扇独我心寒。七弦琴无心弹”

“八行书无处传，九连环从中断，十里长庭望眼欲穿。”

这是一个数着月份逐步完成几个月相思的过程。

1.7 java设计模式笔记【创建模式第七篇】

发表时间: 2009-10-23

原始模型（Prototype）：
属于对象的创建模式。通过给出一个原型对象来指明所要创建的对象的类型，然后用复制这个原型对象的办法创建出更多同类型的对象。

java.lang.Object.clone()方法，所有javaBean都必须实现Cloneable接口，才具有clone功能；

Java语言提供的Cloneable接口只起一个作用，就是在运行时期通知java虚拟机可以安全地在这个类上使用clone()方法。通过调用
这个clone()方法可以得到一个对象的复制。由于Object类本身并实现Cloneable接口，因此如果所考虑的类没有实现Cloneable接
口时，调用clone()方法会抛出CloneNotSupportedException异常。

```
//PandaToClone类
        public class PandaToClone implements Cloneable{
                private int height, weight, age;

                public PandaToClone(int height, int weight){
                        this.age = 0;
                        this.weight = weight;
                        this.height = height;
                }

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

                public int getAge(){
                        return age;
                }
                //其他重复

                //克隆方法
                public Object clone(){
                        //创建一个本类对象并返回给调用者
```

```
                PandaToClone temp = new PandaToClone(height,weight);
                temp.setAge(age);
                return (Object)temp;
        }
}

//客户端
public class Client{
        private static PandaToClone thisPanda, thatPanda;

        public static void main(String args[]){
                thisPanda = new PandaToClone(15,25);
                thisPanda.setAge(3);

                thatPanda = (PandaToClone)thisPanda.clone();

                System.out.println("Age of this panda :" + thisPanda.getAge());
                System.out.println("Height : " + thisPanda.getHeight());
                System.out.println("Weigth : " + this.Panda.getWeight());
                System.out.println("Age of that panda :" + thatPanda.getAge());
                System.out.println("Height : " + thatPanda.getHeight());
                System.out.println("Weigth : " + that.Panda.getWeight());
        }
}
```

二、克隆满足的条件：

1、clone()方法将对象复制了一份并返还给调用者。一般而言，克隆方法满足以下条件：

1）：对任何的对象x，都有：x.clone() != x。换言之，克隆对象与原始对象不是同一个对象。

2）：对任何的对象x，都有：x.clone().getClass() == x.getClass(),换言之，克隆对象与原始对象的类型一样。

3）：如果对象x的equals()方法定义恰当的话，那么x.clone().equals(x)应当是成立的。

三、原始模型模式的结构;

1、简单形式的原始模型模式

1）：这种形式涉及到三个角色：

。 客户（Client）角色：客户类提出创建对象的请求。

。 抽象原始（Prototype）角色：这是一个抽象角色，通常由一个java接口或java抽象类实现。此角色给出所有的具体原始类所需要的接口。

。 具体原型（Concrete Prototype）角色：被复制的对象。此角色需要实现抽象的原型角色所要求的接口。

```
//客户端
                        public class Client{
                                private Prototype prototype;

                                public void operation(Prototype example){
                                        Prototype p = (Prototype)example.clone();
                                }
                        }

                        //抽象原型角色
                        public interface Prototype implements Cloneable{
                                Prototype clone();
                        }

                        //具体原型角色
                        public class ConcretePrototype implements Prototype{
                                //克隆方法
                                public Object clone(){
                                        try{
                                                return super.clone();
                                        }catch(CloneNotSupportedException e){
                                                return null;
                                        }
                                }
                        }
```

2、登记形式的原始模型模式

1）：有如下角色：

。 客户端（Client）角色：客户端类向管理员提出创建对象的请求。

。 抽象原型（Prototype）角色：这是一个抽象角色，通常由一个接口或抽象
此角色给出所有的具体原型类所需要的接口。

。 具体原型（Concrete Prototype）角色：被复制的对象。需要实现抽象原

。 原型管理器（Prototype Manager）角色：创建具体原型类的对象，并记录

```
//抽象原型角色
public interface Prototype inplements Cloneable{
        public Object clone();
}

//具体原型角色
public class ConcretePrototype implements Prototype{
        //克隆方法
        public synchronizer Object clone(){
                Prototype temp = null;
                try{
                        temp  = (Prototype)super.clone();
                        return (Object)temp;
                }catch(CloneNotSupportedException e){
                        System.out.println("Clone failed.");
                }finally{
                        return (Object)temp;
                }
        }
}

//原型管理器角色保持一个聚集，作为多所有原型对象的登记，这个惧色提供必要的方法
  供外界增加新的原型对象和取得已经登记过的原型对象。
//原型管理器类
import java.util.Vector;
public class PrototypeManager{
        private Vector objects = new Vector();

        //聚集管理方法：增加一个新的对象
        public void add(Prototype object){
                objects.add(object);
        }
        //聚集管理方法：取出聚集中的一个对象
        public Prototype get(int i){
                return (Prototype)objects.get(i);
        }
        //聚集管理方法：给出聚集的大小
```

```
                        public int getSize(){
                                return objects.size();
                        }
                }

        //客户端
        public class Client{
                private PrototypeManager mgr;
                private Prototype prototype;
                public void registerPrototype(){
                        prototype = new ConcretePrototype();
                        Prototype copytype = (Prototype)prototype.clone();
                        mgr.add(copytype);
                }
        }
```

3、两种形式的比较

简单形式和登记形式的原始模型模式各有其长处和短处。

1）如果需要创建的原型对象数目较少而且比较固定的话，可以采取第一种形式，也即简单形式的原始模型模式。

在这种情况下，原型对象的引用可以由客户端自己保存。

2）如果要创建的原型对象数目不固定的话，可以采取第二种形式，也即登记形式的原始模型模式。

在这种情况下，客户端并不保存对原型对象的引用，这个任务被交给管理员对象。在复制一个原型对象之前，客户端

可以查看管理员对象是否已经有一个满足要求的原型对象。如果有，可以直接从管理员类取得这个对象引用；如果没有，

客户端就需要自行复制此原型对象。

4、模式的实现

1）：浅复制（浅克隆）

被复制对象的所有变量都含有与原来的对象相同的值，而所有的对其他对象的引用都仍然指向原来的对象，换言之，

浅复制仅仅复制所考虑的对象，而不复制它所引用的对象。

2）：深复制（深克隆）

被复制对象的所有的变量都含有与原来的对象相同的值，除去那些引用其他对象的变量。那些引用其他对象的变量将
指向被复制过的新对象，而不再是原有的那些被引用的对象。换言之，深复制把要复制的对象所引用的对象都复制了
一遍，而这种对被引用到的对象的复制叫做间接复制。

3）：利用串行化来作深复制
把对象写到流里的过程是串行化（Serilization）过程，写到流里的是对象的一个拷贝，而原来对象仍然存在于JVM
里面，因此“腌成咸菜”（串行化）的只是对象的一个拷贝，java咸菜（并行化Deserialization）还可以回鲜。

在java语言里深复制一个对象，常常可以使一个对象实现Serialization接口，然后把对象（实际上只是对象的一个拷贝）
写到一个流里（腌成咸菜），再成流里读回来（把咸菜回鲜），便可以重建对象。

```
//深复制
public Object deepClone(){
//将对象写到流里
ByteArrayOutputStream bo = new ByteArrayOutputStream();
ObjectOutputStream oo = new ObjectOutputStream(bo);
oo.writeObject(this);
//从流里读回来
ByteArrayInputStream bi = new ByteArrayInputStream(bo.toByteArray());
ObjectInputStream oi = new ObjectInputStream(bi);
return (oi.readObject());
}
```

这样做的前提就是对象以及对象内部所有引用到的对象都是可串行化的，否则，就需要仔细考察那些不可串行化的对象可否
设成transient，从而将之排除在复制过程之外。
5、模式的选择
1)：如果客户角色所使用的原型对象只有固定的几个，那么使用单独的变量来引用每一个原型对象就很方便，
这时候简单形式的原始模型模式就比较合适。
2）：而如果客户端所创建的原型对象的个数不是固定的，而是动态地变化的，那么就不妨使用一个Vector类型的变量
用来动态地存储和释放原型对象。这时候，使用第二中形式的原始模型模式就比较合乎需要。

```
//孙大圣本人类
                public class TheGreatestSage{           //GreatestSage（大圣）
                        private Monkey monkey = new Monkey();

                        public void change(){
                                //创建大圣本尊对象
                                Monkey copyMonkey;
                                //空循环一会儿
                                for(int i = 0; i < 2000; i ++){}
                                //克隆大圣本尊
                                copyMonkey = (Monkey)monkey.clone();
                                System.out.println("Monkey King's birth date = " + monkey.getBi
                                System.out.println("Copy monkey's birth date = " + copyMonkey.g
                                System.out.println("Monkey King == Copy Monkey?" + (monkey == c
                                System.out.println("Monkey King's staff == Copy Monkey's staff?
                                (monkey.getStaff() == copyMonkey.getStaff()));
                        }

                        public static void main(String args[]){
                                TheGreatestSage sage = new TheGreatestSage();
                                sage.change();
                        }
                }

                / / / / / / / / / / / / / / / /
                import java.util.Date;

                public class Monkey inplements Colenable{
                        private int height;
                        private int weight;
                        private GoldRingedStaff staff;
                        private Date birthDate;

                        public Monkey(){
                                this.birhtDate = new Date();
                        }
```

```
        public Object clone(){
                Monkey temp = null;
                try{
                        temp  = (Monkey)super.clone();
                }catch(CloneNotSupportedException e){
                        System.out.println("Clone failed.");
                }finally{
                        return (Object)temp;
                }
        }

        public int getHeight(){
                return height;
        }

        public void setHeight(int height){
                this.height = height;
        }
        //其他重复
        //生日的取值方法
        public Date getBirthDate(){
                return birthDate;
        }

        public void setBirthDate(Date birhtDate){
                this.birthDate = birhtDate;
        }
}

//金箍棒类
public class GoldRingedStaff{
        private float height = 100.0F;
        private float diameter = 10.0F;

        public GoldRingedStaff(){
                //write your code here;
        }
```

```
        }

        //增长行为，每次调用长度和半径增加一倍
        public void grow(){
                this.diameter *= 2.0;
                this.height *= 2;
        }
        //缩小行为，每次调用长度和半径减少一半
        public void shrink(){
                this.diameter /= 2;
                this.height /= 2;
        }
        //移动
        public void move(){
                //write you code
        }

        public float getHeight(){
                return height;
        }
        public void setHeight(float heigth){
                this.height = height;
        }
        //半径的取值。赋值方法。。。。。。。

//深复制,必须要实现serializable接口
        import java.util.Date;
        import java.io.IOException;
        import java.lang.ClassNotFoundException;

        public class TheGreatestSage{
                private Monkey monkey = new Monkey();

                public void change() throws IOException,ClassNotFoundException{
                        Monkey copyMonkey;
                        for(int i = 0; i < 2000; i ++){}
                        copyMonkey = (Monkey)monkey.deepClone();
```

```
                System.out.println("Monkey King's birth date = " + monkey.getBi
                System.out.println("Copy monkey's birth date = " + copyMonkey.g
                System.out.println("Monkey King == Copy Monkey?" + (monkey == c
                System.out.println("Monkey King's staff == Copy Monkey's staff?
                (monkey.getStaff() == copyMonkey.getStaff()));
        }

        public static void main(String args[]) throws IOException,ClassNotFound
                TheGreatestSage sage = new TheGreatestSage();
                sage.change();
        }
}

//金箍棒类
import java.util.Date;
import java.io.Serializable;

public class GoleRingedStaff implements Cloneable,Serializable{
        private float height = 100.0F;
        private float diameter = 10.0F;

        public GoldRingedStaff(){
                //write ..............
        }
        //跟上一样
}
```

6、在什么情况下使用原始模型模式

假设一个系统的产品类是动态加载的，而且产品类有一定的等级结构。

这是采取原始模型模式，给每一个产品类配备一个克隆方法（大多数的时候只需要给产品类等级结构的根类配备一个克隆方法）

便可以避免使用工厂模式所带来的具有固定等级结构的工厂类。

7、原始模型模式的优点与缺点

1）：特有的优点

／：原始模型模式原许动态地增加或减少产品类。由于创建产品类实例的方法是产品类内部具有的，因此，增加新产品对

整个结构没有影响。

／：原始模型模式提供简化的创建结构。工厂方法模式常常需要有一个与产品类等级结构相同的等级结构，

而原始模型模式就不需要这样。

／：具有给一个应用软件动态加载新功能的能力。例如，一个分析Web服务器的记录文件的应用软件，针对每一种记录文件

格式，都可以由一个相应的“格式类”负责。如果出现了应用软件所不支持的新的Web服务器，只需要提供一个格式类

的克隆，并在客户端登记即可，而不必给每个软件的用户提供一个全新的软件包。

／：产品类不需要非得有任何事先确定的等级结构，因为原始模型模式适合用于任何的等级结构。

2）：主要缺点

／：每一个类都必须配备一个克隆方法，配备克隆方法需要对类的功能进行通盘考虑，这对于全新的类来说不是很难，

而对于已经有的类不一定很容易，特别是当一个类引用不支持串行化的间接对象，或者引用含有循环结构的时候。

8、原始模型模式与其他模式的关系

1）原始模型模式与合成模式的关系

原始模型模式经常与合成模式一同使用，因为原型对象经常是合成对象。

2）原始模型模式与抽象工厂模式的区别

如果系统不需要动态地改变原型对象，抽象工厂模式可以成为原始模型模式的替代品

3）原始模型模式与门面（Facade）模式的区别

原型模型模式的客户端通常可以将系统的其他对象与参与原始模型模式的对象分割开，起到一个门面对象的作用。

4）原始模型模式与工厂犯法尬，模式的关系

如果原型对象只有一种，而且不会增加的话，工厂方法模式可以成为一种替代模式。

5）原始模型模式与装饰（Decorator）模式的关系

原始模型模式常常与装饰模式一同使用。

1.8 java设计模式笔记【创建模式第八篇】

发表时间: 2009-10-23

预定式存储：
为了保证在任何情况下键值都不会出现重复，应当使用预定式键值存储办法。在请求一个键值时，首先将数据库中的键值更新为下一个可用值，
然后将旧值提供给客户端。这样万一出现运行中断的话，最多就是这个键值被浪费掉。
记录式存储：
键值首先被返还给客户端，然后记录到数据库中。这样做的缺点是，一旦系统中断，就有可能出现客户端已经使用了一个键值，而这个键值却
没有来得及存储到数据库中的情况。在系统重启后，系统还会从这个已经被使用过的键值开始，从而导致错误。

单例模式应用：序列键管理器

方案一：没有数据库的情况

```
        //序列键生成器(单例类)
        public class KeyGenerator{
                private static KeyGenerator keygen = new KeyGenerator();
                private int key = 1000;

                private KeyGenerator(){}

                //静态工厂方法，提供自己的实例
                public static KeyGenerator getInstance(){
                        return keygen;
                }
                //取值方法,提供一个合适的键值
                public synchronized int getNextKey(){
                        return key ++;
                }
        }

        //客户端
        public class Client{
                private static KeyGenerator keygen;
```

```
        public static void main(String args[]){
                keygen = KeyGenerator.getInstance();
                System.out.println("Key = " + keygen.getNextKey());
                System.out.println("Key = " + keygen.getNextKey());
                System.out.println("Key = " + keygen.getNextKey());
        }
}
//当系统重启时，计数就要重新开始
```

方案二：有数据库的情况：

```
//序列键生成器(单例类)
public class KeyGenerator{
        private static KeyGenerator keygen = new KeyGenerator();
        private int key = 1000;

        private KeyGenerator(){}

        //静态工厂方法，提供自己的实例
        public static KeyGenerator getInstance(){
                return keygen;
        }
        //取值方法,提供一个合适的键值
        public synchronized int getNextKey(){
                return getNextKeyFromDB();
        }

        private int getNextKeyFromDB(){
                String sql1 = "UPDATE KeyTable SET keyValue = keyValue + 1";
                String sql2 = "SELECT keyValue FROM KeyTable";
                //execute the update SQL
                //run the select query
                //示意性的返还一个数值
                return 1000;
        }
}
//每次都向数据库查询键值，将这个键值登记到表里，然后将查询结果返还给客户端
```

方案三：键值的缓存方案

```
        //序列键生成器(单例类)
        public class KeyGenerator{
                private static KeyGenerator keygen = new KeyGenerator();
                private static final int POOL_SIZE = 20;                //用于缓存20条记录
                private KeyInfo key;

                private KeyGenerator(){
                        key = new KeyInfo(POOL_SIZE);
                }

                //静态工厂方法，提供自己的实例
                public static KeyGenerator getInstance(){
                        return keygen;
                }
                //取值方法,提供一个合适的键值
                public synchronized int getNextKey(){
                        return key.getNextKey();
                }
        }

        //KeyInfo类
        public class KeyInfo{
                private int keyMax;
                private int keyMin;
                private int nextKey;
                private int poolSize;

                public KeyInfo(int poolSize){
                        this.poolSize = poolSize;
                        retrieveFromDB();                               //retrieve（检索）
                }

                //取值方法，提供键的最大值
                public int getKeyMax(){
                        return keyMax;
                }
```

```
        public int getKeyMin(){
                return keyMin;
        }

        //取值方法，提供键的当前值
        public int getNextKey(){
                if(nextKey > keyMax){
                        retrieveFromDB();
                }
                return nextKey ++;
        }

        //内部方法，从数据库提取键的当前值
        private void retrieveFromDB(){
                String sql1 = "UPDATE KeyTable SET keyValue = keyValue +" + poolSize +
                "WHERE keyName = 'PO_NUMBER'";

                String sql2 = "SELECT keyValue FROM KeyTable WHERE keyName = 'PO_NUMBEF
                //execute the above queries in a transaction and commit it
                //assume the value returned is 1000
                int keyFromDB = 1000;
                keyMax = keyFromDB;
                keyMin = keyFromDB - poolSize + 1;
                nextKey = keyMin;
        }
}

//客户端
public class Client{
        private static KeyGenerator keygen;

        public static void main(String args[]){
                keygen = KeyGenerator.getInstance();
                for(int i = 0; i < 20; i ++){
                        System.out.println("Key(" + (i+1) + ")=" + keygen.getNextKey()
                }
```

```
                }
        }
```

方案四：有缓存的多序列键生成器

```
        //KeyGenerator
        import java.util.HashMap;

        public class KeyGenerator{
                private static KeyGenerator keygen = new KeyGenerator();
                private static final int POOL_SIZE = 20;
                private HashMap keyList = new HashMap(10);

                private KeyGenerator(){

                }

                public static KeyGenerator getInstance(){
                        return keygen;
                }

                public synchronized int getNextKey(String keyName){
                        KeyInfo keyInfo;
                        if(keyList.containsKey(keyName)){
                                keyInfo = (KeyInfo)keyList.get(keyName);
                                System.out.println("Key found");
                        }else{
                                keyInfo = new KeyInfo(POOL_SIZE,keyName);
                                keyList.put(keyName,keyInfo);
                                System.out.println("New key created");
                        }
                        return keyInfo.getNextKey(keyName);
                }
        }

        //KeyInfo类
        public class KeyInfo{
                private int keyMax;
```

```
        private int keyMin;
        private int nextKey;
        private int poolSize;
        private String keyName;

        public KeyInfo(int poolSize,String keyName){
                this.poolSize = poolSize;
                this.keyName = keyName;
                retrieveFromDB();                                   //retrieve(检索)
        }

        //取值方法，提供键的最大值
        public int getKeyMax(){
                return keyMax;
        }

        public int getKeyMin(){
                return keyMin;
        }

        //取值方法，提供键的当前值
        public int getNextKey(){
                if(nextKey > keyMax){
                        retrieveFromDB();
                }
                return nextKey ++;
        }

        //内部方法，从数据库提取键的当前值
        private void retrieveFromDB(){
                String sql1 = "UPDATE KeyTable SET keyValue = keyValue +" + poolSize +
                "WHERE keyName = " + keyName + "";

                String sql2 = "SELECT keyValue FROM KeyTable WHERE keyName = " + keyNAm
                //execute the above queries in a transaction and commit it
                //assume the value returned is 1000
                int keyFromDB = 1000;
```

```
                keyMax = keyFromDB;
                keyMin = keyFromDB - poolSize + 1;
                nextKey = keyMin;
        }
    }

    //客户端
    public class Client{
        private static KeyGenerator keygen;

        public static void main(String args[]){
            keygen = KeyGenerator.getInstance();
            for(int i = 0; i < 20; i ++){
                System.out.println("Key(" + (i+1) + ")=" + keygen.getNextKey("P
            }
        }
    }273P
```

方案五：多例模式

```
    //KeyGennerator
    import java.util.HashMap;
    public class KeyGenerator{
        private static HashMap Kengens = new HashMap(10);
        private static final int POOL_SIZE = 20;
        private KeyInfo keyinfo;

        private KeyGenerator(){}

        private KeyGenerator(String keyName){
            keyinfo = new KeyInfo(POOL_SIZE,keyName);
        }

        public static synchronized KeyGenerator getInstance(String keyName){
            KeyGenerator keygen;
            if(kengens.containsKey(keyName)){
                keygen = (KeyGenerator)kengens.get(keyName);
            }else{
```

```
                        keygen = new KeyGenerator(keyName);
                }
                return keygen;
        }

        public synchronized int getNextKey(){
                return keyinfo.getNextKey();
        }
        //KeyInfo类与方案四里的相同
}

//客户端
public class Client{
        private static KeyGenerator keygen;

        public static void main(String args[]){
                keygen = KeyGenerator.getInstance("PO_NUMBER");
                for(int i = 0; i < 20; i ++){
                        System.out.println("Key(" + (i+1) + ")=" + keygen.getNextKey());
                }
        }
}
```

1.9 java设计模式笔记【结构模式第一篇】

发表时间: 2009-10-23

代理（Proxy）模式
是对象的结构模式，代理模式给某一个对象提供一个代理对象，并由代理对象控制对原对象的引用。

一、代理的种类：
如果按照使用目的来划分，代理有以下几种：
1、远程（Remote）代理：为一个位于不同的地址空间的对象提供一个局域代表对象。这个不同的地址空间可以是在本机器中，
　也可以在另一台机器中。远程代理又叫做大使（Ambassador）。
2、虚拟（Virtual）代理：根据需要创建一个资源消耗较大的对象，使得此对象只在需要时才会被真正创建。
3、Copy - on - Write代理：虚拟代理的一种，把复制（克隆）拖延到只有在客户端需要时，才真正采取行动。
4、保护（Protect or Access）代理：控制对一个对象的访问，如果需要，可以给不同的用户提供不同级别的使用权限。
5、Cache（高速缓存）代理：为某一个目标操作的结果提供临时的存储空间，以便多个客户端可以共享这些结果。
6、防火墙（Firewall）代理：保护目标，不让恶意用户接近。
7、同步化（Synchronization）代理：使几个用户能够同时使用一个对象而没有冲突。
8、智能引用（Smart Reference）代理：当一个对象被引用时，提供一些额外的操作，比如将对此对象调用的次数记录下来。
二、代理模式的结构：
代理模式所涉及的角色有：
1、抽象主题角色：声明了真实主题和代理主题的共同接口，这样一来在任何可以使用真实主题的地方都可以使用代理主题

2、代理主题角色：代理主题角色内部含有对真实主题的引用，从而可以在任何时候操作真实主题对象；
　代理主题角色提供一个与真实主题角色相同的接口，以便可以在任何时候都可以替代真实主体；
　控制对真实主题的引用，负责在需要的时候创建真实主题对象（和删除真实主题对象）；
　代理角色通常在将客户端调用传递给真实的主题之前或者之后，都要执行某个操作，而不是单纯地将调用传递给真实主题对象

3、真实主题角色：定义了代理角色所代表的真实对象。

```
//抽象主题角色
                        public abstract class Subject{
```

```
        public abstract void request();
}

//真实主题角色
public class RealSubject extends Subject{
        public RealSubject(){}

        public void request(){
                System.out.println("From real subject.");
        }
}

//代理主题角色
public class ProxySubject extends Subject{
        private RealSubject realSubject;

        public ProxySubject(){}

        public void request(){
                preRequest();
                if(realSubject == null){
                        realSubject = new RealSubject();
                }
                realSubject.request();
                postRequest();
        }

        private void preRequest(){
                //.................
        }

        private void postRequest(){
                //.................
        }
}

//VectorProxy
```

```
import java.lang.reflect.InvocationHandler;
import java.lang.reflect.Proxy;
import java.lang.reflect.Method;
import java.util.Vector;
import java.util.List;

public class VectorProxy implements InvocationHandler{
        private Object proxyobj;

        public VectorProxy(Object obj){
                proxyobj = obj;
        }

        //静态工厂方法
        public static Object factory(Object obj){
                Class cls = obj.getClass();
                return cls.getClassLoader();
                cls.getInterface(new VectoryProxy(obj));
        }

        public Object invoke(Object proxy, Method method, Object[] args
                System.out.println("before calling" + method);
                if(args != null){
                        for(int i = 0; i < args.length; i ++){
                                System.out.println(args[i] + " ");
                        }
                }
                Object o = method.invoke(proxyobj,args);
                System.out.println("after calling" + method);
                return o;
        }

        public static void main(String args[]){
                List v = null;
                v = (List)factory(new Vector(10));
                v.add("New");
                v.add("York");
```

```
                }
            }
```

三、代理模式的优、缺点

1、优点

1 - 远程代理）优点是系统可以将网络的细节隐藏起来，使得客户端不必考虑网络的存在。

客户完全可以认为代理的对象是局域的而不是远程的，而代理对象承担了大部分的网络通信工作。

2 - 虚拟代理）优点是代理对象可以在必要的时候才将被代理的对象加载。

代理可以对加载的过程加以必要的优化。当一个模块的加载十分耗费资源的时候，虚拟代理的优点就非常明显。

3 - 保护代理）优点是它可以在运行的时间对用户的有关权限进行检查，然后在核实后决定将被调用传递给被代理的对象。

4 - 智能引用代理）在访问一个对象时可以执行一些内务处理操作。

```
//一个实例
        //此实例调用的次序是：
                1、客户端调用代理对象
                2、代理对象调用AccessValidator对象，确定用户确实具有相应的权限
                3、代理对象调用RealSearcher对象，完成查询功能
                4、代理对象调用UsageLogger对象，完成计数功能
                5、代理对象将RealSearcher对象的查询结果返回给客户端。
        //客户端
        public class Client{
                //声明一个静态类型为Searcher的静态变量
                private static Searcher searcher;

                public static void main(String args[]){
                        //此静态变量的真实类型为Proxy
                        searcher = new Proxy();
                        String userId = "Admin";
                        String searchType = "SEARCH_BY_ACCOUNT_NUMBER";
```

```
                String result = searcher.doSearch(userId,searchType);
                System.out.println(result);
        }
}

//抽象主题角色
public interface Searcher{
        String doSearch(String userId,String searchType);
}

//代理角色
public class Proxy implements Searcher{
        private RealSearcher searcher;
        private UsageLogger usageLogger;
        private AccessValidator accessValidator;

        public Proxy(){
                searcher = new RealSearcher();
        }

        public String doSearch(String userId, String keyValue){
                if(checkAccess(userId)){
                        String result = searcher.doSearch(null,keyValue);
                        logUsage(userId);
                        return result;
                }else{
                        return null;
                }
        }

        public boolean checkAccess(String userId){
                accessValidator = new AccessValidator();
                return accessValidator.validateUser(userId);
        }

        private void logUsage(String userId){
                UsageLogger logger = new UsageLogger();
```

```
                logger.setUserId(userId);
                logger.save();
        }
}

//真实主题
public class RealSearcher implements Searcher{
        public RealSearcher(){}

        //真实的查询工作在这里进行
        public String doSearch(String userId,String keyValue){
                String sql = "select * from data_table where key_col = '" + keyValue +
                //execute SQL Statement
                //concatenate a result String
                return "result set";
        }
}

//检查权限的对象
public class AccessValidator{
        //用户检查发生在这里
        public boolean validatorUser(String userId){
                if(userId.equals("Admin")){
                        return true;
                }else{
                        false;
                }
        }
}

//记录次数
public class UsageLogger{
        private String userId;

        public void setUserId(String userId){
                this.userId = userId;
        }
```

```
        public void save(){
                String sql = "insert into usage_table(user_id)" + "values(" + userId +
                //execute sql
        }

        public void save(*String userId){
                this.userId = userId;
                save();
        }
}
```

1.10 java设计模式笔记【结构模式第二篇】

发表时间: 2009-10-23

合成（Composite）模型模式：
属于对象的结构模式，有时又叫做部分 - 整体（Part - Whole）模式。
合成模式将对象组织到树结构中，可以用来描述整体与部分的关系。合成模式可以使客户端将单纯元素与复合元素同等看待。

1、模式涉及到的三个角色
1）抽象构件（Component）角色：这是一个抽象角色，它给参加组合的对象规定一个接口。这个角色给出共有的接口机器默认行为。
2）树叶构件（Leaf）角色：代表参加组合的树叶对象。一个树叶没有下级的子对象。定义出参加组合的原始对象的行为。
3）树枝构件（Composite）角色：代表参加组合的有子对象的对象，并给出树枝构件对象的行为。

2、合成模式的实现根据所实现接口的区别分为两种形式，分别称为安全式和透明式。虽然这是模式的实现问题，但是由于它影响到模式结构的细节
合成模式可以不提供对服对象的管理方法，但是合成模式必须在合适的地方提供子对象的管理方法。

1）透明方式
作为第一种选择，在Compotent里面声明所有的用来管理子类对象的方法，包括add()、remove()、以及getChild()方法。
这样做的好处是所有的构件类都有相同的接口。在客户端看来，树叶类对象与合成类对象的区别起码在接口层次上消失了，客户端
可以同等地对待所有的对象。这就是透明形式的合成模式。

这个选择的缺点是不够安全，因为树叶对象和组合类对象的在本质上是有区别的。树叶类对象不可能有下一个层次的对象，
因此add() remove() getChild()方法没有意义，但是在编译时期不会出错，而只会在运行期出错。

2）安全方式
第二种选择是在Composite类里而声明所有的用来管理子类对象的方法。这样的做法是安全的做法。
因为树叶类型的对象根本就没有管理子类对象的方法，因此，如果客户端对树叶类对象使用这些方法时，程序会在编译时期出错。
编译通不过，就不会出现运行时期错误

这个方式的缺点就是不够透明，因为树叶类和合成类将具有不同的接口。

（安全式）这种形式涉及到三个角色;

1）抽象构件角色：这是一个抽象角色，它给参加组合的对象定义出公共的接口及其默认行为，可以用来管理所有的子对象。

合成对象通常把它所包含的子对象当作类型为Component的对象。

在安全式的合成模式里，构件角色并不定义出管理子对象的方法，这一定义由树枝构件对象给出。

2）树叶构件角色：树叶对象是没有下级子对象的对象，定义出参加组合的原始对象的行为。

3）树枝构件角色：代表参加组合的有下级子对象的对象。树枝构件类给出所有的管理子对象的方法，如add() remove()

components()的声明。

抽象构件角色由一个java接口实现，它给出两个对所有的子类均有用的方法：getComposite() sampleOperation()

```
//抽象构件角色Component的接口
public interface Component{
        //返还自己的实例
        Composite getComposite();
        //某个商业方法
        void sampleOperation();
}

//树枝构件（Composite）类
import java.util.Vector;
import java.util.Enumeration;

public class Composite implements Component{
        private Vector componentVector = new Vector();

        //返还自己的实例
        public Composite getComposite(){
                return this;
        }

        public void sampleOperation(){
                Enumeration enumeration = getChild();   //components()
                while(enumeration.hasMoreElement()){
                        ((Component)enumeration.nextElement()).sampleOperation(
                }
```

```
        }

        //聚集管理方法
        public void add(Component component){
                componentVector.addElement(component);
        }

        public void remove(Component component){
                componentVector.removeElement(component);
        }

        //聚集管理方法，返还聚集的Enumeration对象
        public Enumeration getChild(){
                return componentVector.elements();
        }
}

//树叶（Leaf）构件类
import java.util.Enumeration;

public class Leaf implements Component{
        public void sampleOperation(){
                //.................
        }

        //返还自己的实例
        public Composite getComposite(){
                //.............
                return null;
        }
}
```

（透明式）这种形式涉及到三个角色：

1） 抽象构件角色：这是一个抽象角色，它给参加组合的对象规定一个就口，规范共有的接口及默
管理所有的子对象，要提供一个接口以规范取得和管理下层组件的接口，包括add() remove(

2） 树叶构件角色：代表参加组合的树叶对象，定义出参加组合的原始对象的行为。树叶类会给出

之类的用来管理子类对象的方法的平庸实现。

3） 树枝构件角色：代表参加组合的有子对象的对象，定义出这样的对象的行为。

```
//抽象构件角色接口
import java.util.Enumeration;
public interface Component{
        void sampleOperation();

        //返还自己的实例
        Composite getComposite();

        //聚集管理方法
        void add(Component component);

        void remove(Component component);

        Enumeration getChild();
}

//树枝构件类
import java.util.Vector;
import java.util.Enumeration;

public class Composite implements Component{
        private Vector componentVector = new Vector();

        public Composite getComposite(){
                return this;
        }

        public void sampleOperation(){
                Enumeration enumeration = getChild();
                while(enumeration.hasMoreElement()){
                        ((Component)enumeration.nextElement()).sampleOperation(
                }
        }
```

```
        public void add(Component component){
                componentVector.addElement(component);
        }

        public void remove(Component component){
                componentVector.removeElement(component);
        }

        public Enumeration getChild(){
                return componentVector.elements();
        }
}

import java.util.Enumeration;

public class Leaf implements Component{
        public void sampleOperation(){
                //..................
        }

        public void add(Component component){}

        public void remove(Component component){}

        public Composite getComposite(){
                return null;
        }

        public Enumeration getChild(){
                return null;
        }
}
```

3、实现合成模式时，有几个可以考虑的问题：

1）明显地给出父类对象的引用。在子对象里面给出父对象的引用，这样可以很容易地遍历所有的父对象，管理

合成结构。

有了这个引用，可以很方便地应用责任链模式。

2）在通常的系统里，可以使用享元模式实现构件的共享，但是由于合成模式的对象经常要有对父类的引用，因此共享不

容易实现

3）抽象构件类应当多“重”才算好。

4）有时候系统需要遍历过一个树枝构件的子构件很多次，这时候就可以把遍历子构件的结果暂时存放在父构件里面，作为缓存。

5）使用什么数据类型来存储子对象。可以使用Vector或者其他数组的聚集。

6）Composite向子类的委派。客户端不应当直接调用树叶类，应当由其父类向树叶类进行委派。这样可以增加代码的复用性

```
//抽象构件角色的实现
          public abstract class Graphics{
               public abstract void draw();
          }

          //树枝构件角色（Picture）
          import java.util.Vector;

          public class Picture extends Graphics{
               private Vector list = new Vector(10);

               public void draw(){
                       for(int i = 0; i < list.size(); i ++){
                              Graphics g = (Graphics)list.get(i);
                              g.draw();
                       }
               }

               public void add(Graphics g){
                       list.add(g);
               }

               public void remove(Graphics g){
                       list.remove(s);
               }
```

```
        public void getChild(int i){
                return (Graphics)list.get(i);
        }
}

//树叶构件角色（Line）
public class Line extends Graphics{
        public void draw(){
                //............
        }
}

//树叶构件角色（Rectangle）
public class Rectangle extends Graphics{
        public void draw(){
                //.............
        }
}

//树叶构件角色（Circle）
public class Circle extends Graphics{
        public void draw(){
                //.............
        }
}
```

4、在下面的情况下应当考虑使用合成模式

1）需要描述对象的部分和整体的等级结构。

2）需要客户端忽略掉个体构件和组合构件的区别。客户端必须平等对待所有的构件，包括个体构件和组合构件。

5、合成模式的优缺点

1）优点：合成模式可以很容易地增加新种类的构件。

使用合成模式可以使客户端变得很容易设计，因为客户端不需要知道构件是树叶构件还是树枝构件。

2）缺点：使用合成模式后，控制树枝构件的类型就不太容易。

用继承的方法来增加新的行为很困难

1.11 java设计模式笔记【结构模式第三篇】

发表时间: 2009-10-23

亨元（Flyweight Pattern）模式
1、亨元模式的用意
亨元模式是对象的结构模式。亨元模式以共享的方式高效地支持大量的细粒度对象。
亨元模式能做到共享的关键是区分内蕴状态和外蕴状态

一个内蕴状态是存储在亨元对象内部的，并且是不会随环境改变而有所不同的。因此，一个亨元可以具有内蕴状态并可以共享。

一个外蕴状态是随环境改变而改变的、不可以共享的状态。亨元对象的外蕴状态必须由客户端保存，并在亨元对象被创建之后，
在需要使用的时候再传入到亨元对象内部。

外蕴状态不可以影响亨元对象的内蕴状态的，它们是相互独立的。

2、亨元模式的种类
根据所涉及的亨元对象的北部表象，亨元模式可以分为单纯亨元模式和复合亨元模式两种形式。

3、亨元模式的实现：
1）单纯亨元模式涉及的角色
1 - 抽象亨元角色：此角色是所有的具体亨元类的超类，为这些规定出需要实现的公共接口，那些需要外蕴状态的操作
　可以通过方法的参数传入。抽象亨元的接口使得亨元变得可能，但是并不强制子类实行共享，因此并非所有的亨元
　对象都是可以共享的
2 - 具体亨元角色：实现抽象亨元角色所规定的接口。如果有内蕴状态的话，必须负责为内蕴状态提供存储空间。
　亨元对象的内蕴状态必须与对象所处的周围环境无关，从而使得亨元对象可以在系统内共享。有时候具体亨元角色
　又叫做单纯具体亨元角色，因为复合亨元角色是由单纯具体亨元角色通过复合而成的
3 - 复合亨元角色：复合亨元角色所代表的对象是不可以共享的，但是一个复合亨元对象可以分解成为多个本身是单纯亨元
　对象的组合。复合亨元角色又称做不可共享的亨元对象。
4 - 亨元工厂角色：本角色负责创建和管理亨元角色。本角色必须保证亨元对象可以被系统适当地共享。
　当一个客户端对象请求一个亨元对象的时候，亨元工厂角色需要检查系统中是否已经有一个符合要求的亨元对

象，

如果已经有了，享元工厂角色就应当提供这个已有的享元对象；如果系统中没有一个适当的享元对象的话，享元工厂角色就应当创建一个新的合适的享元对象。

5 - 客户端角色：本角色还需要自行存储所有享元对象的外蕴状态。

```
//抽象享元角色
                        public abstract class Flyweight{
                                public abstract void operation(String state);
                        }

                        //具体享元角色
                        具体享元角色的主要责任：
                                1）实现了抽象享元角色所声明的接口，也就是operation()方法。operation
                                2）为内蕴状态提供存储空间，在本实现中就是intrinsicState属性。享元模
                                   这里的内蕴状态是Character类型，是为了给符合享元的内蕴状态选做Str
                        public class ConcreteFlyweight extends Flyweight{
                                private Character intrinsicState = null;

                                public ConcreteFlyweight(Character state){
                                        this.intrinsicState = state;
                                }

                                //外蕴状态作为参量传入到方法中
                                public void operation(String state){
                                        System.out.print("\nInternal State = " + intrinsicState
                                         state);
                                }
                        }

                        //具体复合享元角色
                        具体复合享元角色的责任：
                                1）复合享元对象是由单纯的享元对象通过复合而成，因此它提供了add()这样
                                   由于一个复合享元对象具有不同的聚集元素，这些聚集元素在复合享元对象
                                   享元对象的状态是会改变的，因此复合享元对象是不能共享的。
                                2）复合享元角色实现了抽象享元角色所规定的接口， 也就是operation()方
                                   代表复合享元对象的外蕴状态，。一个复合享元对象的所有单纯享元对象元
                                   外蕴状态相等的，而一个复合享元对象所含有的单纯享元对象的内蕴状态一
```

```
import java.util.Map;
import java.util.HashMap;
import java.util.Iterator;

public class ConcreteCompositeFlyweight extends Flyweight{
        private HashMap flies = new HashMap(10);
        private Flyweight flyweight;

        public ConcreteCompositeFlyweight(){}

        //增加一个新的单纯享元对象到聚集中
        public void add(Character key, Flyweight fly){
                flies.put(key,fly);
        }

        //外蕴状态作为参量传入到方法中
        public void operation(String extrinsicState){
                Flyweight fly = null;
                for(Iterator it = flies.entrySet().iterator()); it.hasN
                        Map.Entry e = (Map.Entry)it.next();
                        fly = (Flyweight)e.getValue();
                        fly.operation(extrinsicState);
                }

        }
}

//享元工厂角色
import java.util.Map;
import java.util.HashMap;
import java.util.Iterator;

public class FlyweightFactory{
        private HashMap flies = new HashMap();

        public FlyweightFactory(){}
```

```
//复合享元工厂方法，所需状态以参量形式传入，这个参量恰好可以使用Stri
public Flyweight factory(String compositeState){
        ConcreteCompositeFlyweight compositeFly = new ConcreteC
        int length = compositeState.length();
        Character state = null;
        for(int i = 0; i < length; i ++){
                state = new Character(compositeState.charAt(i))
                System.out.println("factory(" + state +")");
                compositeFly.add(state,this.factory(state));
        }
        return compositeFly;
}

//单纯享元工厂方法
public Flyweight factory(Character state){
        //检查具有此状态的享元是否已经存在
        if(flies.containsKey(state)){
                //具有此状态的享元已经存在，因此直接将它返回
                retun (Flyweight)flies.get(state);
        }else{
                //具有此状态的享元不存在，因此创建新实例
                Flyweight fly = new ConcreteFlyweight(state);
                //将实例存储到聚集中
                flies.put(state,fly);
                //将实例返回
                return fly;
        }
}

public void checkFlyweight(){
        Flyweight fly;
        int i = 0;
        System.out.println("\n==========CheckFlyweight()=======
        for(Iterator it = flies.entrySet().iterator(); it.hasNe
                Map.Entry e = (Map.Entry) it.next();
                System.out.println("Item" + (++i) + ";" + e.get
```

```
                                }
                                System.out.println("\n==========CheckFlyweight()=======
                        }
                }
```

4、模式的实现

1）使用不变模式实现享元角色

亨元模式里的亨元对象不一定非得是不变对象，但是很多的亨元对象确实被设计成了不变对象。

由于不变对象的状态之后就不再变化，因此不变对象满足亨元模式对亨元对象的要求。

2）使用备忘录模式实现亨元工厂角色

亨元工厂负责维护一个表，通过这个表把很多全同的实例与代表它们的一个对象联系起来。这就是备忘录模式的应用。

3）使用单例模式实现亨元工厂角色

系统往往只需要一个亨元工厂的实例，所以亨元工厂可以设计成为单例模式。

在单纯的共享模式中使用单例模式实现共享工厂角色

```
                import java.util.Map;
                import java.util.HashMap;
                import java.util.Iterator;
                public class FlyweightFactorySingleton{
                        private HashMap flies = new HashMap();
                        private static FlyweightFactorySingleton myself = new FlyweightFactoryS

                        private FlyweightFactorySingleton(){}

                        public static FlyweightFactorySingleton getInstance(){
                                return myself;
                        }

                        //工厂方法，向外界提供含有指定内蕴状态的对象
                        public synchronized Flyweight factory(Character state){
                                //检查具有此状态的亨元是否已经存在
```

```
                if(flies.containsKey(state)){
                        //具有此状态的享元对象已经存在，因此直接返还此对象
                        return (Flyweight)flies.get(state);
                }else{
                        Flyweight fly = new ConcreteFlyweight(state);
                        flies.put(state,fly);
                        return fly
                }
        }

        //辅助方法，打印所有已经创建的享元对象清单
        public void checkFlyweight(){
                Flyweight fly;
                int i = 0;
                System.out.println("\n===========checkFlyweight==============='
                for(Iterator it = flies.entrySet().iterator(); it.hasNext();){
                        Map.Entry e = (Map.Entry)it.next();
                        System.out.println("Item" + (++i) + ":" + e.getKey());
                }
                System.out.println("\n===========checkFlyweight==============='
        }

}

//客户端代码
public class ClientSingleton{
        private static FlyweightFactorySingleton factory;

        public static void main(String args[]){
                factory = FlyweightFactorySingleton.getInstance();
                Flyweight fly = factory.factory(new Character('a'));
                fly.operation("First Call");
                fly = factory.factory(new Character('b'));
                fly.operation("Second call");
                Flyweight fly = factory.factory(new Character('a'));
                fly.operation("Third Call");
                factory.checkFlyweight();
```

```
        }
}

//将一个共享工厂角色用单例模式实现
import java.util.Map;
import java.util.HashMap;
import java.util.Iterator;

public class FlyweightFactorySingleton{
        private static FlyweightFactorySingleton myself = new FlyweightFactoryS
        private HashMap flies = new HashMap();
        private Flyweight inkFlyweight;

        private FlyweightFactorySingleton(){}

        public static FlyweightFactorySingleton getInstance(){
                return new FlyweightFactorySingleton();
        }

        public Flyweight factory(String complexState){
                ConcreteCompositeFlyweight complexFly = new ConcreteCompositeFl
                int length = complexState.length();
                Character state = null;

                for(int i = 0; i < length; i ++){
                        state = new Character(complexState.charAt(i));
                        System.out.println("factory(" + state + ")");
                        complexFly.add(state,this.factory(state));
                }
                return complexFly;
        }

        public synchronized Flyweight factory(Character state){
                //检查具有此状态的享元是否已经存在
                if(flies.containsKey(state)){
                        return (Flyweight)flies.get(state);
```

```
                }else{
                        Flyweight fly = new ConcreteFlyweight(state);

                        flies.put(state,fly);
                        return fly;
                }
        }

        public void checkFlyweight(){
                Flyweight fly;
                int i = 0;
                System.out.println("\n===========checkFlyweight==============
                for(Iterator it = flies.entrySet().iterator(); it.hasNext();){
                        Map.Entry e = (Map.Entry)it.next();
                        System.out.println("Item" + (++i) + ":" + e.getKey());
                }
                System.out.println("\n===========checkFlyweight==============
        }
    }

    //一个应用亨元模式的咖啡摊例子

    //抽象亨元角色,serve()它没有参量是因为没有外蕴状态
    public abstract class Order{
        //将咖啡卖客人
        public abstract void serve();
        //返还咖啡的名字
        public abstract String getFlavor();
    }


//具体亨元角色
public class Flavor extends Order{
        private String flavor;

        public Flavor(String flavor){
                this.flavor = flavor;
```

```
        }

        public String getFlavor(){
                return this.flavor;
        }

        //将咖啡卖给客人
        public void serve(){
                System.out.println(System.out.println("Serving flavor " + flavo
        }
}

//工厂角色
public class FlavorFactory{
        private Order[] flavors = new Flavor[10];
        private int ordersMade = 0;
        private int totalFlavors = 0;

        //工厂方法，根据所需的风味提供咖啡
        public Order getOrder(String flavorToGet){
                if(ordersMade > 0){
                        for(int i  = 0 ; i < ordersMade; i ++){
                                if(flavorToGet.equals(flavors[i].getFlavor())){
                                        return flavors[i];
                                }
                        }
                }
                flavors[ordersMade] = new Flavor(flavorToGet);
                totalFlavors++;
                return flavors[ordersMade++];
        }

        //辅助方法，返还创建过的风味对象的个数
        public int getTotalFlavorsMade(){
                return totalFlavors;
        }
}
```

```
//客户端代码，代表咖啡摊侍者
public class ClientFlavor{
        private static Order[] flavors = new Flavor[20];
        private static int ordersMade = 0;
        private static FlavorFactory flavorFactory;

        //提供一杯咖啡
        private static void takeOrders(String aFlavor){
                flavors[ordersMade++] = flavorFactory.getOrder(aFlavor);
        }

        public static void main(String args[]){
                flavorFactory = new FlavorFactory();

                takeOrders("Black Coffee");
                takeOrders("Capucino");
                takeOrders("Espresso");
                takeOrders("Espresso");
                takeOrders("Capucino");
                takeOrders("Capucino");
                takeOrders("Black Coffee");
                takeOrders("Espresso");
                takeOrders("Capucino");

                //将所创建的对象卖给客人
                for(int i = 0; i < ordersMade; i ++){
                        flavors[i].serve();
                }
                System.out.println("\nTotal teaFlavor objects made: " +
                                                        flavorFactor.getTotalFlavorsMad
        }
}

//如果这个例子再大点，可以增加一个外蕴状态，也就是说可以增加桌子号，建立一个桌子类，将
参数传给serve(Table table).
```

5、亨元模式的使用情况

当以下所有的条件都满足时，可以考虑使用亨元模式：

（1）一个系统有大量的对象。

（2）这些对象耗费大量的内存

（3）这些对象的状态中大部分都可以外部化

（4）这些对象可以按照内蕴状态分成很多的组，当把外蕴对象从对象中剔除时，每一个组都可以仅用一个对象代替。

（5）软件系统不依赖于这些对象的身份，换言之，这些对象可以是不可分辨的。

应当在有足够多的亨元实例可供共享时才值得使用亨元模式。

6、怎样做到共享

一个亨元对象之所以可以被很多的用户端共享，是因为它只含有可以共享的状态，而没有不可以共享的状态，这就是使用亨元模式的前提。

要做到符合亨元模式这一点，需要分两步走：

（1）将可以共享的状态和不可以共享的状态从此常规类中区分开来，将不可共享的状态从类里剔除出去。

那些对所有客户端都去相同的值的状态是可以共享的状态；而那些对不同的客户端有不同值的状态是不可以共享的状态

（2）这个类的创建过程必须由一个工厂对象加以控制。

这个工厂对象应当使用一个内部列表保存所有的已经创建出来的对象。当客户端请求一个新的对象时，

工厂对象首先检查列表，看是否已经有一个对象。如果已经有了，就直接返还此对象，如果没有就创建一个新对象。

亨元模式要求将可以共享的状态设置为内蕴状态，而将不可以共享的状态设置成外蕴状态，将它们外部化。

7、亨元模式的优缺点

亨元模式的优点在于它大幅度降低内存中对象的数量。但是，它做到这一点所付出的代价也是很高的：

（1）亨元模式使得系统更加复杂。为了使对象可以共享，需要将一些状态外部化，这使得程序的逻辑复杂化

（2）亨元模式将亨元对象的状态外部化，而读取外部状态使得运行时间稍微变长。

1.12 java设计模式笔记【结构模式第四篇】

发表时间: 2009-10-23

门面（Facade）模式：
是对象的结构模式。外部与一个子系统的通信必须通过一个统一的门面对象进行。549P

一、门面模式的角色：
1、门面角色：客户端可以调用这个角色的方法。此角色知晓相关的（一个或多个）子系统的功能和责任。
在正常情况下，本角色会将所有从客户端发来的请求委派到相应的子系统去。

2、子系统角色：可以同时有一个或者多个子系统。每一个子系统都不是一个单独的类，而是一个类的集合。
每一个子系统都可以被客户端直接调用，或者被门面角色调用。子系统并不知道门面的存在，对于子系统而言，
门面仅仅是另外一个客户端而已。

二、一个使用门面模式的保安系统

```
//客户端
                public class Client{
                        private static SecurityFacade security;

                        private static void main(String args[]){
                                security.activate();
                        }
                }

                //门面类
                public class SecurityFacade{
                        private Camera camera1 , camera2;
                        private Light light1, light2, light3;
                        private Sensor sensor;
                        private Alarm alarm;

                        public void activate(){
                                camera1.turnOn();
                                camera2.turnOn();
                                light1.turnOn();
```

```
                light2.turnOn();
                light3.turnOn();
                sensor.activate();
                alarm.activate();
        }

        public void deactivate(){
                camera1.turnOff();
                camera2.turnOff();
                light1.turnOff();
                light2.turnOff();
                light3.turnOff();
                sensor.deactivate();
                alarm.deactivate();
        }
}

//录像机
public class Camera{
        public void turnOn(){
                System.out.println("Turning on the camera.");
        }

        public void turnOff(){
                System.out.println(Turning off the camera.);
        }

        //转动录像机
        public void rotate(int degrees){
                System.out.println("rorating the camera by " + degrees + " degr
        }
}

//灯
public class Light{
        public void turnOn(){
                System.out.println("Turning on the light.");
```

```
        }

        public void turnOff(){
                System.out.println("Turning off the light.");
        }

        //换灯泡
        public void changeBulb(){
                System.out.println("changing the light-bulb.");
        }
}

//感应器
public class Sensor{
        public void activate(){
                System.out.println("Acticating the sensor.");
        }

        public void deactivate(){
                System.out.println("Deactivating the sensor.");
        }

        //触发感应器
        public void trigger(){
                System.out.println("The sensor has been triggered.");
        }
}

public class Alarm{
        public void activate(){
                System.out.println("Activating the alarm.");
        }

        public void deactivate(){
                System.out.println("Deactivate the alarm.");
        }
```

```
        //拉响警报器
        public void ring(){
                System.out.println("Ring the alarm.");
        }

        //停掉警报器
        public void stopRing(){
                System.out.println("Stop the alarm.");
        }
}
```

1.13 java设计模式笔记【结构模式第五篇】

发表时间: 2009-10-23

桥梁（Bridge）模式：

一、桥梁模式的用意

桥梁模式的用意是：将抽象化与实现化脱耦，使得二者可以独立地变化。

1、抽象化

存在于多个实体中的共同的概念性联系，就是抽象化。作为一个过程，抽象化就是忽略一些信息，从而把不同的实体当做同样

的实体对待。

2、实现化

抽象化给出的具体实现就是实现化。

一个类的实例就是这个类的实现化，一个具体子类是它的抽象超类的实现化。而在更加复杂的情况下，实现化也可以是与抽象化

等级结构相平行的等级结构，同样可以由抽象类和具体类组成。

3、脱耦

所谓耦合，就是两个实体的行为的某种强关联。而将它们的强关联去掉，就是脱耦。

在这里，脱耦是指将抽象化和实现化之间的耦合解脱开，或者说是将它们之间的强关联改换成弱关联。

所谓强关联，就是在编译时期已经确定的，无法在运行时期动态改变的关联；

所谓弱关联，就是可以动态地确定并且可以在运行时期动态改变的关联。

在java中，继承关系是强关联，而聚合关系是弱关联。

因此，桥梁模式中的所谓脱耦，就是指在一个软件系统的抽象化和实现化之间使用组合／聚合关系而不是继承关系，

从而使两者可以相对独立地变化。这就是桥梁模式的用意。

二、一个制造飞机的系统例子

```
//抽象化角色类
                public abstract class Airplace{
                        public abstract void fly();
                        protected AirplaceMaker airplaneMaker;
                }

                //修正化抽象角色（载客飞机）
                public class PassengerPlane extends Airplane{
                        public void fly(){
```

```
                //..........
        }
}

//修正化抽象角色（载货飞机）
public class CargoPlane extends Airplane{
        public void fly(){
                //..............
        }
}

//实现化角色(飞机制造商)
public abstract class AirplaneMaker{
        public abstract void produce();
}

//具体实现化类
public class Airbus extends AirplaneMaker{
        public void produce(){
                //..........
        }
}

public class Boeing extends AirplaneMaker{
        public void produce(){
                //..........
        }
}

public class MD extends AirplaneMaker{
        public void produce(){
                //..........
        }
}
```

三、在什么情况下使用桥梁模式

1、如果一个系统需要在构件的抽象化角色和具体化角色之间增加更多的灵活性，避免在两个层次之间建立静态的联系。

2、设计要求实现化角色的任何改变不应当影响客户端，或者说实现化角色的改变对客户端是完全透明的。

3、一个构件有多于一个的抽象化角色和实现化角色，系统需要它们之间进行动态耦合。

4、虽然在系统中使用继承是没有问题的，但是由于抽象化角色和具体化角色需要独立变化，设计要求需要独立管理这两者。

1.14 java设计模式笔记【结构模式第六篇】

发表时间: 2009-10-23

适配器模式（Adapter Pattern）（另称 - 变压器模式）：
把一个类的接口变换成客户端所期待的另一种接口，从而使原本因接口不匹配而无法在一起工作的两个类能够在一起工作

1、(类适配器)模式所涉及的角色有:
1 / 目标（Target）角色：这就是所期待得到的接口。由于是类适配器模式，因此目标不可以是类。
2 / 源（Adaptee）角色：现有需要适配的接口。
3 / 适配器（Adapter）角色：适配器类是本模式的核心。适配器把源接口转换成目标接口。显然，这一角色不可以是接口，
　而必须是具体类。

```
//目标角色类
                public interface Target{
                        //源类有的方法
                        void sampleOperation1();
                        //源类没有的方法
                        void sampleOperation2();
                }
                源类（具体类）
                public class Adaptee{
                        //源类含有的方法sampleOperation1()
                        public void sampleOperation1(){}
                }
                //适配器角色
                public class Adapter extends Adaptee implements Target{
                        public void sampleOperation2(){}
                }
```

2、（对象适配器）
1）：模式所涉及的角色有：
1 / 目标（Target）角色：这就是所期待的接口，目标可以是具体的或抽象的类
2 / 源（Adaptee）角色：现有需要适配的接口
3 / 适配器（Adapter）角色：适配器类是本模式的核心。适配器把源接口转换成目标接口，这一角色必须是具体类

```
//Target类
                        public interface Target{
                                //源类有的方法
                                void sampleOperation1();
                                //源来没有的方法
                                void sampleOperation2();
                        }
                        源类（具体类）
                        public class Adaptee{
                                //源类含有的方法sampleOperation1()
                                public void sampleOperation1(){}
                        }
                        //适配器类
                        public class Adapter implements Target{
                                private Adaptee adaptee;

                                public Adapter(Adaptee adaptee){
                                        super();
                                        this.adaptee = adaptee;
                                }
                                //源类有的方法，适配器直接委派就可以了
                                public void sampleOperation1(){
                                        adaptee.sampleOperation();
                                }
                                //源类没有，需要补充
                                public void sampleOperation2(){
                                        //.............
                                }
                        }
```

3、适配器模式的用意是将接口不同而功能相同或者相近的两个接口加以转换，这里面包括适配器角色补充了一个源角色没有的方法。

4、对象适配器模式的效果

1）一个适配器可以把多种不同的源适配到同一个目标，换言之，同一个适配器可以把源类和它的子类都适配到目标接口。

2）与类的适配器模式相比，要想置换源类的方法就不容易。如果一定要置换掉源类的一个或多个方法，就只好先做一个源

类的子类，将源类的方法置换掉，然后再把原来的子类当做真正的源进行适配。

3）虽然要想置换源类的方法不容易，但是要想增加一些新的方法则方便的很，而且新增加的方法可同时适用于所有的源。

5、在什么情况下使用适配器模式

1）系统需要使用现有的类，而此类的接口不符合系统的需要。

2）想要建立一个可以重复使用的类，用于与一些彼此之间没有太大关联的一些类，包括一些可能在将来引进的类一起工作。

这些源类不一定有很复杂的接口。

3）（对对象适配器模式而言）在设计里，需要改变多个已有的子类的接口，如果使用类的适配器模式，就要针对每一个子类做

一个适配器，而这不太实际

```
//Itermeration类
                import java.util.Iterator;
                import java.util.*;
                import java.util.Enumeration;

                public class Itermeration implements Enumeration{
                        private Iterator it;

                        public Itermeration(Iterator it){
                                this.it = it;

                                //是否存在下一个元素
                                public boolean hasMoreElements(){
                                        return it.hasNext();
                                }

                                //返还下一个元素
                                public Object nextElement() throws NoSuchElementException{
                                        return it.next();
                                }
                        }
                }
```

```
//Enuterator类
import java.util.Iterator;
import java.util.*;
import java.util.Enumeration;

public class Enuterator implements Iterator{
        Enumeration enum;

        public Enuterator(Enumeration enum){
                this.enum = enum;
        }

        //是否存在下一个元素
        public boolean hasNext(){
                return enum.hasMoreElements();
        }

        //返还下一个元素
        public Object next() throws NoSuchElementsException{
                return enum.nextElement();
        }

        //删除当前的元素（不支持）
        public void remove(){
                throw new UnsupportedOperationException();
        }
}
 - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

//立方体类
public class Cube{
        private double width;

        public Cube(double width){
                this.width = width;
        }
```

```
        //计算体积
        public double calculateVolume(){
                return width*width*width;
        }

        //计算面积
        public double calculateFaceArea(){
                return width*width;
        }

        //长度的取值方法
        public double getWidth(){
                return this.width;
        }

        //长度的赋值方法
        public void setWidth(double width){
                this.width = width;
        }
}

//目标接口角色
public interface BallIF{
        //计算面积
        double calculateVolume();
        //半径的取值方法
        double getRadius();
        //半径的赋值方法
        void setRadius(double radius);
}

//适配器类角色
public class MagicFinger implements BallIF{
        private double redius = 0;
        private static final double PI = 3.14D;
        private Cube adaptee;
```

```
        public MagicFinger(Cube adaptee){
                super();
                this.adaptee = adaptee;
                radius = adpatee.getWidth();
        }

        //计算面积
        public double calculateArea(){
                return PI*4.0D*(radius);
        }

        public double calculateVolume(){
                return PI*(4.0D/3.0D)*(radius*radius*radius);
        }

        //半径取值方法
        public double getRadius(){
                return radius;
        }

        public void setRadius(double radius){
                this.radius = radius;
        }
}
```

6、本模式在实现的时候有以下这些值得注意的地方;

1) 目标接口可以省略。此时，目标接口和源接口实际上是相同的。由于源是一个接口，而适配器类是一个类（或抽象类）

因此这种做法看似平庸而并并平庸，它可以使客户端不必实现不需要的方法。

2）适配器类可以是抽象类，这可以在缺省适配情况下看到。

3）带参数的适配器模式。使用这种方法可以根据参数返还一个合适的实例给客户端

7、适配器模式与其他模式的关系

1）适配器模式与桥梁模式的关系

桥梁模式的用意是要把实现和它的接口分开，以便它们可以独立地变化。桥梁模式并不是用来把一个已有的对象接到不相

匹配的接口上的。当一个客户端只知道一个特定的接口，但是有必须与具有不同接口的类打交道时，就应当使用适配器模式。

2）适配器模式与装饰模式的关系

一个装饰类也是位于客户端和另外一个Compontent对象之间的，

在它接到客户端的调用后把调用传给一个或几个Component对象。

一个纯粹的装饰类必须与Compotent对象在接口上的完全相同，并增强后者的功能。

与适配器类不同的是，装饰类不能改变它所装饰的Compotent对象的接口。

3）适配器模式与缺省适配模式的关系

1.15 java设计模式笔记【结构模式第七篇】

发表时间: 2009-10-23

装饰（Decorator）模式:又名包装模式（425P）
装饰模式以对客户端透明的方式扩展的功能，是继承关系的一个替代方案。

装饰模式以对客户透明的方式动态地给一个对象附加上更多的责任。换言之，客户端并不会觉得对象在装饰之前和装饰之后有什么不同。
装饰模式可以在不使用创造更多子类的情况下，将对象的功能加以扩展。

装饰模式使用原来被装饰的类的一个子类的实例，把客户端的调用委派到被装饰类。装饰模式的关键在于这种扩展是完全透明的。

装饰模式中的各个角色有：
1、抽象构件角色：给出一个抽象接口，以规范准备接收附加责任的对象。
2、具体构件角色：定义一个将要接收附加责任的类。
3、装饰角色：持有一个构件对象的实例，并定义一个与抽象接口一致的接口。
4、具体装饰角色：负责给构件对象“贴上”附加责任。

```
//抽象构件角色
                public interface Component{
                        //商业方法
                        void sampleOperation();
                }

                //装饰角色
                public class Decorator implements Component{
                        private Component component;

                        public Decorator(Component component){
                                this.component = component;
                        }

                        public Decorator(){
                                //...........
                        }
```

```
        //商业方法，委派给构件 ]
        public void sampleOperation(){
                component.sampleOperation();
        }
}
//接口的实现方法值得注意，每一个实现的方法都委派给父类，但并不是单纯地委派，而是有功能

//具体构件类
public class ConcreteComponent implements Component{
        public ConcreteComponent(){
                //.........
        }

        public void sampleOperation(){
                //.............
        }
}

//具体装饰类
public class ConcreteDecorator extends Decorator{
        public void sampleOperation(){
                super.sampleOperation();
        }
}
```

装饰模式应当在什么情况下使用；

1、需要扩展一个类的功能，或给一个类增加附加责任。

2、需要动态地给一个对象增加功能，这些功能可以再动态地撤销。

3、需要增加由一些基本功能的排列组合而产生的非常大量的功能，从而使继承关系变得不现实。

```
/ / / / / / / / / / / / / / / / / / / / / / / / / / / / / / / / / / / / / / / /
//大圣接口
public interface 齐天大圣{
        public 齐天大圣();
        public void move();
}
```

```
//大圣的七十二般变化，抽象装饰类
public class 七十二般变化 implements 齐天大圣{
        private 齐天大圣 c;
        public 七十二般变化(齐天大圣 c){
                super();
                this.c = c;
        }

        public void move(){
                c.move();
        }
}

//大圣本尊类，具体构件角色
public class 大圣本尊 extends 齐天大圣{
        public 大圣本尊(){
                //代码
        }

        public void move(){
                //代码
        }
}

//大圣的变化类，具体装饰类
public class 鱼 extends 七十二般变化{
        public 鱼(){
                //代码
        }

        public 鱼(齐天大圣 c){
                //........
        }

        public void move(){
                super.move();
```

```
        }
    }

    //实现
    齐天大圣 c = new 大圣本尊();
    齐天大圣 fish = new 鱼(c);

    从而系统把大圣从一只猴子装饰成了一条鱼（也就是把鱼的功能加到了猴子身上）
```

使用装饰模式的优点和缺点

1、优点

1）：装饰模式与继承关系的目的都是要扩展对象的功能，但是装饰模式可以提供比继承更多的灵活性。

装饰模式允许系统动态地决定“贴上”一个需要的“装饰”，或者除掉一个不需要的“装饰”。

继承关系则不同，继承关系是静态的，它在系统运行前就决定了。

2）：通过使用不同的具体装饰类以及这些装饰类的排列组合，设计师可以创造出很多不同行为的组合。

3）：这种比继承更加灵活机动的特性，也同时意味着装饰模式比继承更加易于出错。

2、缺点

1）：由于使用了装饰模式，可以比使用继承关系需要较少数目的类。使用较少的类，当然使设计比较易于进行。但是

在另一方面，使用装饰模式会产生比使用继承关系更多的对象。更多的对象会使得查错变得困难，特别是这些对象

看上去都很相像。

```
透明性的要求：指出一个重要的实现问题，通常叫做针对抽象编程。装饰模式对客户端的透明性要求程序不要声明一个
        类型，而应当声明一个Component类型的变量。

    //GrepView类，搜索某文件里某个词出现的地方，并打印出来
    import java.io.*;

       public class GrepView {
            PrintStream out;

            public GrepView(){
                out = System.out;
            }
```

```
                public void println(String line){
                        out.println(line);
                }
        }

//GrepReader类
import java.io.*;

public class GrepReader extends FilterReader{
        protected String substring;

        protected BufferedReader in;
        private int lineNumber;

        public GrepReader(FileReader in, String substring){
                super(in);
                this.in = new BufferedReader(in);
                this.substring = substring;
                lineNumber = 0;
        }

        public final String readLine() throws IOException{
                String line;

                do{
                        line = in.readLine();
                        lineNumber ++;
                }while((line != null) && line.indexOf(substring) == -1);

                return line;
        }

        public int getLineNo(){
                return lineNumber;
        }
}
```

```
//Grep类
import java.io.*;

public class Grep {
        static GrepReader g;
        private static GrepView gv = new GrepView();

        public static void main(String args[]){
        String line;
        if(args.length <= 1){
                gv.println("Usage:java Grep");
                gv.println("      no regexp");
                gv.println("      files to be searched in");
                System.exit(1);
        }
        try{
                gv.println("\nGrep: 搜索 " + args[0] + "文件" + args[1]);
                gv.println("文件和行号\n");
                g = new GrepReader(new FileReader(args[1]),args[0]);
                for(;;){
                        line = g.readLine();
                        if(line == null)break;
                        gv.println(args[1] + ":" + g.getLineNo() + ":\t" + line
                }
                g.close();
        }catch(IOException e){
                gv.println(e.getMessage());
        }
}
```

//一个发票系统

要求：打印发票，发票分为三个部分

1、发票头部：上面有顾客的名字，销售的日期。

2、发票主部：销售的货物清单，包括商品名字、购买的数量、单价、小计。

3、发票的尾部：商品总金额。

在这个系统里使用装饰模式，那么发票的头部和尾部都可以分别由具体装饰类HeaderDecorator和具体装饰类FooterDecorator

有多少种头部和尾部，就可以有多少种相应的具体装饰类。

其中SalesOrder代表发票的主部，Order是抽象构件角色，OrderDecorator是抽象装饰角色。OrderLine是一个发票的货品清单中的

一行，给出产品名、产品单价、所购单位数、小计金额。Order类有一个Vector聚集，用以存储任意多个OrderLine对象

```
//抽象构件角色类
                import java.util.*;
                import java.text.NumberFormat;

                public abstract class Order{
                        private OrderLine lnkOrderLine;
                        protected String customerName;  //customer用户，顾客
                        protected Date salesDate;
                        protected Vector items = new Vector(10);

                        public void print(){
                                for(int i = 0; i < items.size(); i ++){
                                        OrderLine item = (OrderLine)items.get(i);
                                        item.printLine();
                                }
                        }

                        //客户名的取值方法
                        public String getCustomerName(){
                                return customerName;
                        }

                        public void setCustomerName(String customerName){
                                this.customerName = customerName;
                        }

                        //销售日期的取值方法
```

```
                public Date getSalesDate(){
                        return salesDate;
                }

                public void setSalesDate(Date salesDate){
                        this.salesDate = salesDate;
                }

                //增加一行销售产品
                public void addItem(OrderLine item){
                        items.add(item);
                }

                public void removeItem(OrderLine item){
                        items.remove(item);
                }

                //返还总额
                public double getGrandTotal(){
                        double amount = 0.0D;
                        for(int i = 0; i < items.size(); i ++){
                                OrderLine item = (OrderLine)items.get(i);
                                amount += item.getSubtotal();
                        }
                        return amount;
                }

                //工具方法将金额格式化
                protected String formatCurrency(double amount){
                        return NumberFormat.getCurrencyInstance().format(amount);
                }

        }

//具体构件类
public class SalesOrder extends Order{
        public SalesOrder(){}
```

```
        public void print(){
                super.print();
        }
}

//抽象装饰角色
public abstract class OrderDecorator extends Order{
        protected Order order;

        public OrderDecorator(Order order){
                this.order = order;
                this.setSalesDate(order.getSalesDate());
                this.setCustomerName(order.getCustomerName());
        }

        public void print(){
                super.print();
        }
}

//具体装饰类
public class HeaderDecorator extends OrderDecorator{
        public HeaderDecorator(Order anOrder){
                super(anOrder);
        }

        public void print(){
                this.printHeader();
                super.order.print();
        }

        private void printHeader(){
                System.out.println("\t***\tI N V O I C E\t***");
                System.out.println("XYZ Incorporated\nDate of Sale:");
                System.out.println(order.getSalesDate());
                System.out.println("=======================");
```

```
                System.out.println("Item\t\tUnits\tUnit Price\tSubtotal");
        }
}

//具体装饰类
import java.text.NumberFormat;

public class FooterDecorator extends OrderDecorator{
        public FooterDecorator(Order anOrder){
                super(anOrder);
        }

        public void print(){
                super.order.print();
                printFooter();
        }

        public void printFooter(){
                System.out.println("============================");
                System.out.println("Total\t\t\t" + formatCurrency(super.order.getGrandT
        }
}

//OrderLine类
import java.text.NumberFormat;
public class OrderLine{
        private String itemName;
        private int units;
        private double unitPrice;

        //产品名字取值方法
        public String getItemName(){
                return itemName;
        }

        public void setItemName(String itemName){
                this.itemName = itemName;
```

```
        }

        //单位数量的取值方法
        public int getUnits(){
                return units;
        }

        public void setUnits(int units){
                this.units = units;
        }

        //单价的取值方法
        public double getUnitPrice(){
                return unitPrice;
        }

        public void setUnitPrice(double unitPrice){
                this.unitPrice = unitPrice;
        }

        public void printLine(){
                System.out.println(itemName + "\t" + units + "\t" + formatCurrency(unit
                formatCurrency(getSubtotal()));
        }

        //小计金额取值方法
        public double getSubtotal(){
                return unitPrice * units;
        }

        //工具方法，将金额格式化
        private String formatCurrency(double amount){
                return NumberFormat.getCurrencyInstance().format(amount);
        }

}
```

```
//客户端
import java.util.*;

public class Client{
        private static Order order;

        public static void main(String args[]){
                order = new SalesOrder();
                order.setSalesDate(new Date());
                order.setCustomerName("XYZ Repair Shop");
                OrderLine line1 = new OrderLine();
                line1.setItemName("FileWheel Tire");
                line1.setUnitPrice(154.23);
                line1.setUnits(4);
                order.addItem(line1);
                OrderLine line2 = new OrderLine();
                line2.setItemName("Front Fender");
                line2.setUnitPrice(300.45);
                line2.setUnits(1);
                order.addItem(line2);

                order = new HeaderDecorator(new FooterDecorator(order));
                order.print();
        }
}
```

（InputStream）

1、 原始流处理器

1）ByteArrayInputStream：为多线程的通信提供缓冲区操作功能，接收一个Byte数组作为流的源。

2）FileInputStream：建立一个与文件有关的输入流。接收一个File对象作为流的源。

3）PipedInputStream：可以与PipedOutputStream配合使用，用于读入一个数据管道的数据。接收一个PipedOutputStream作为源。

4）StringBufferInputStream：将一个字符串缓冲区转换为一个输入流。接收一个String对象作为流的源。

2、链接流处理器

所谓链接流处理器，就是可以接收另一个（同种类的）流对象（也就是前面所说的链接流源）作为流源，并对之进行功能扩展的类。

InputStream类型的链接流处理器包括以下几种，它们都接收另一个InputStream对象作为流源。

1）FileInputStream：称为过滤流，它将另一个输入流作为流源。这个类的子类包括以下几种：

（1）BufferedInputStream：用来从硬盘将数据读入到一个内存缓冲区中，并从此缓冲区提供数据。

（2）DataInputStream：提供基于多字节的读取方法，可以读取原始数据类型的数据。

（3）LineNumberInputStream：提供带有行计数功能的过滤输入流。

（4）PushbackInputStream：提供特殊的功能，可以将已经读取的字节“推回”到输入流中。

2）ObjectInputStream：可以将使用ObjectOutputStream 串行化的原始数据类型和对象重新并行化。

3）Sequence(顺序，序列)InputStream：可以将两个已有的输入流连接起来，形成一个输入流，从而将多个输入流排列构成一个输入流序列。

3、装饰模式的角色：

1）抽象构件角色：由InputStream扮演。这是一个抽象类，为各种子类型流处理器提供统一的接口。

2）具体构件角色：由ByteArrayInputStream、FileInputStream、PipedInputStream以及StringBufferInputStream等流 处理器扮演。

他们实现了抽象构件角色所规定的接口，可以被链接流处理器所装饰。

3）抽象装饰角色：由FileterInputStream扮演。它实现了InputStream所规定的接口。

4）具体装饰角色：由几个类扮演，分别是DataInputStream、BufferedInputStream以及两个不常用到的类LineNumberInputStream和

PushbackInputStream。

OutputStream跟上面差不多

1.16 java设计模式笔记【结构模式第八篇】

发表时间: 2009-10-23

XMLProperties与适配器模式举例：

```
//- - - - -
        import java.io.FileReader;
        import org.xml.sax.XMLReader;
        import org.xml.sax.InputSource;
        import org.xml.sax.helpers.XMLReaderFactory;
        import org.xml.sax.helpers.DefaultHandler;

        public class MySAXApp extends DefaultHandler {
        public MySAXApp() {
                super();
        }

        public void startDocument(){
                System.out.println("start document()!");
        }

        public void character(char[] ch, int start, int end){
                System.out.println("character()");
                for(int i = start; i < start + end; i ++){
                        System.out.println(ch[i]);
                }
        }

        public void endDocument(){
                System.out.println("end document()!");
        }

        public static void main(String args[]) throws Exception {
                XMLReader xr = XMLReaderFactory.createXMLReader();
                MySAXApp handler = new MySAXApp();
                xr.setContentHandler(handler);
                FileReader fr = new FileReader("/home/briup/work/xml_jd0810/src/day01/student.x
```

```
        xr.parse(new InputSource(fr));
    }
    }



    //适配器类XMLProperties类
    import org.xml.sax.DocumentHandler;
    import org.xml.sax.SAXException;
    import org.xml.sax.Locator;
    import org.xml.sax.AttributeList;
    import org.xml.sax.Parser;
    import org.xml.sax.InputSource;
    import java.io.IOException;
    import java.io.OutputStream;
    import java.io.InputStream;
    import java.io.BufferedInputStream;
    import java.io.FileInputStream;
    import java.io.File;
    import java.io.PrintWriter;
    import java.util.Properties;
    import java.util.Enumeration;

    //此类是一个Properties数据和XML数据之间的转换器

    public class XMLProperties extends Properties{
        XMLParser p = null;

        //从一个输入流读入XML
        public synchronized void load(InputStream in) throws IOException{
            try{
                p = new XMLParser(in, this);
            }catch(SAXException e){
                throw new IOException(e.getMessage());
            }
        }
```

```
//将XML文件读入
public synchronized void load(File file) throws SAXException,IOException{
        InputStream in = new BufferedInputStream(new FileInputStream(file));
        XMLParser p = null;
        try{
                p = new XMLParser(in, this);
        }catch(SAXException e){
                System.out.println(e);
                throw e;
        }
}

//调用store(OutputStream out, String header)方法
public synchronized void save(OutputStream out, String header){
        try{
                store(out, header);
        }catch(IOException ex){
                ex.printStackTrace();
        }
}

//将此property列表写入到输出流里
public synchronized void store(OutputStream out, String header) throws IOExcept
        PrintWriter wout = new PrintWriter(out);
        wout.println("<?xml version="1.0"?>");
        if(header != null){
                wout.println("<!-- " + header + "-->");
        }
        wout.print("<properties>");
        for(Enumeration e = keys(); e.hasMoreElements();){
                String key = (String)e.nextElement();
                String value = (String)get(key);
                wout.print("\n<key name = \>"" + key + "\">");
                wout.print(encode(value));
                wout.print("</key>");
        }
        wont.print("\n</properties>");
```

```
                wout.flush();
        }

        public StringBuffer encode(String string){
                int len = string.length();
                StringBuffer buffer = new StringBuffer(len);
                char c;
                for(int i = 0; i < len; i ++){
                        switch(c == string.charAt(i)){
                                case '&':
                                        buffer.append("&amp;");
                                        break;
                                case '><':
                                        buffer.append("<");
                                        break;
                                case '"':
                                        buffer.append(">");
                                        break;
                                default:
                                        buffer.append(c);
                        }
                }
                return buffer;
        }

        //创建一个没有默认内容的空的property
        public XMLProerties(){
                super();
        }

        //创建一个空的property,并以传入的property为默认值
        public XMLProperties(Properties defaults){
                super(defaults);
        }
}
```

1.17 java设计模式笔记【行为模式第一篇】

发表时间: 2009-10-23

备忘录（Memento Pattern）模式
备忘录模式又叫做快照模式（Snapshot Pattern）或Token模式，是对象的行为模式。

备忘录对象是一个用来存储另外一个对象内部状态的快照的对象。备忘录模式的用意是在不破坏封装的条件下，将一个对象的状态捕捉住，并外部化
存储起来，从而可以在将来合适的时候把这个对象还原到存储起来的状态。备忘录模式常常与命令模式和迭代子模式一同使用。

常见的软件系统往往不止存储一个状态，而是需要存储多个状态。这些状态常常是一个对象历史发展的不同阶段的快照，存储这些快照的备忘录对象
叫做此对象的历史，某一个快照所处的位置叫做检查点。

备忘录角色：
备忘录角色有如下的责任。
1、将发起人（Originator）对象的内部状态存储起来，备忘录可以根据发起人对象的判断来决定存储多少
发起人（Originator）对象的内部状态。
2、备忘录可以保护其内容不被发起人对象之外的任何对象所读取。备忘录有两个等效的接口：
1、窄接口：负责人（Caretaker）对象（和其他除发起人对象之外的任何对象）看到的是备忘录的窄
接（narrow interface)，这个窄接口只允许它把备忘录对象传给其他的对象；

2、宽接口：与负责人对象看到的窄接口相反的是，发起人对象可以看到一个宽接口（wide interface），
这个宽接口允许它读取所有的数据，以便根据数据恢复这个发起人对象的内部状态。853P

发起人角色：
发起人角色有如下责任：
1、创建一个含有当前的内部状态的备忘录对象。
2、使用备忘录对象存储其内部状态。

负责人角色：
负责人角色有如下的责任：
1、负责保存备忘录对象
2、不检查备忘录对象的内容。

宽接口和白箱：

发起人角色

```
public class Originator{
        private String state;

        //工厂方法，返还一个新的备忘录对象
        public Memento createMemento(){
                return new Memento(state);
        }

        //将发起人恢复到备忘录对象所记载的状态
        public void restoreMemento(Memento memento){
                this.state = memento.getState();
        }

        //状态的取值方法
        public String getState(){
                return this.state;
        }

        //状态的赋值方法
        public void setState(String state){
                this.state = state;
                System.out.println("Current state = " + this.state);
        }
}
```

备忘录模式要求备忘录对象提供两个不同的接口：一个宽接口提供给发起人对象，另一个窄接口提
宽接口允许发起人读取到所有的数据；窄接口只允许它把备忘录对象传给其他的对象而看不到内部

```
//备忘录角色
public class Memento{
        private String state;

        public Memento(String state){
                this.state = state;
        }
```

```
        public String getState(){
                return this.state;
        }

        public void setState(String state){
                this.state = state;
        }
}
```

负责人角色负责保存备忘录对象，但是从不修改（甚至不查看）备忘录对象的内容（一个更好的实
对象中读取个修改其内容）

```
//负责人角色
public class Caretaker{
        private Memento memento;

        //备忘录的取值方法
        public Memento retrieveMemento(){
                return this.memento;
        }

        //备忘录的赋值方法
        public void saveMemento(Memento memento){
                this.memento = memento;
        }
}

//客户端
public class Client{
        private static Originator o = new Originator();
        private static Caretaker c= new Caretaker();
        private static void main(String[] args){
                //该负责人对象的状态
                o.setState("On");
                //创建备忘录对象，并将发起人对象的状态存储起来
                c.saveMemento(o.createMemento());
                //修改发起人对象的状态
```

```
                o.setState("Off");
                //恢复发起人对象的状态
                o.restoreMemento(c.retrieveMemento());
        }
    }
```

首先将发起人对象的状态设置成“On”（或者任何有效状态），并且创建一个备忘录对象将这个状的状态改成“Off”（或者任何状态）；最后又将发起人对象恢复到备忘录对象所存储起来的状态，存储的任何状态）

备忘录系统运行的时序是这样的：

（1）将发起人对象的状态设置成“On”。

（2）调用发起人角色的createMemento()方法，创建一个备忘录对象将这个状态存储起

（3）将备忘录对象存储到负责人对象中去。

备忘录系统恢复的时序是这样的：

（1）将发起人对象的状态设置成“Off”；

（2）将备忘录对象从负责人对象中取出；

（3）将发起人对象恢复到备忘录对象所存储起来的状态，“On”状态。

白箱实现的优缺点

白箱实现的一个明显的好处是比较简单，因此常常用做教学目的。白箱实现的一个明显的

窄接口或者黑箱实现

```
    //发起人角色
    public class Originator{
        private String state;

        public Originator(){
        }

        //工厂方法，返还一个新的备忘录对象
        public MementoIF createMemento(){
            return new Memento(this.state);
        }

        //将发起人恢复到备忘录对象记录的状态
        public void restoreMemento(MementoIF memento){
            Memento aMemento = (Memento)memento;
```

```
                this.setState(aMemento.getState());
        }

        public String getState(){
                return this.state;
        }

        public void setState(){
                this.state = state;
                System.out.println("state = " + state);
        }

        protected class Memento implements MementoIF{
                private String savedState;
                private Mememto(String someState){
                        savedState = someState;
                }

                private void setState(String someState){
                        savedState = someState;
                }

                private String getState(){
                        return savedState;
                }
        }
}

public interface MementoIF{}

public class Caretaker{
        private MementoIF memento;

        public MementoIF retrieveMemento(){
                return this.memento;
        }
```

```
        public void saveMemento(MementoIF memento){
                this.memento = memento;
        }
}

public class Client{
        private static Originator o = new Originator();
        private static Caretaker c = new Caretaker();

        public static void main(String args[]){
                //改变负责人对象的状态
                o.setState("On");
                //创建备忘录对象，并将发起人对象的状态存储起来
                c.saveMemento(o.createMemento());
                //修改发起人对象的状态
                o.setState("Off");
                //恢复发起人对象的状态
                o.restoreMemento(c.retrieveMemento());
        }
}
```

黑箱实现运行时的时序为;

(1)将发起人对象的状态设置成“On”。

（2）调用发起人角色的 createMemento()方法，创建一个备忘录对象将这个状态存储起

（3）将备忘录对象存储到负责人对象中去。由于负责人对象拿到的仅是 MementoIF类型

恢复时的时序为：

（1）将发起人对象的状态设置成“Off”；

（2）将备忘录对象从负责人对象中取出。注意此时仅能得到 MementoIF接口，因此无法

（3）将发起人对象的状态恢复成备忘录对象所存储起来的状态，，由于发起人对象的内部

这个内部类是传入的备忘录对象的真实类型，因此发起人对象可以利用内部类Meme

存储多个状态的备忘录模式：

```
//发起人角色
import java.util.Vector;
import java.util.Enumeration;

public class Originator{
```

```
        private Vector states;
        private int index;

        public Originator(){
                states = new Vector();
                index = 0;
        }

        public Memento createMementor(){
                return new Mementor(states,index);
        }

        public void restoreMementor(Mementor memento){
                states = memento.getStates();
                index = memento.getIndex ( ) ;
        }

        public void setState(String state){
                this.states.addElement(state);
                index ++;
        }

        //辅助方法，打印出所有的状态
        public void printStates(){
                System.out.println("Total number of states: " + index);
                for(Enumeration e = states.elements();e.hasMoreElements();){
                        system.out.println(e.nextElement());
                }
        }
}

//备忘录角色
import java.util.Vector;

public class Memento{
        private Vector states;
        private int index;
```

```
                public Memento(Vector states,int index){
                        this.states = (Vector)states.clone();
                        this.index = index;
                }

                //状态取值方法
                Vector getStates(){
                        return states;
                }

                //检查点取值方法
                int getIndex(){
                        return this.index;
                }
        }
******************备忘录的构造子克隆了传入的states，然后将克隆存入到备忘录对象内部，这是一个重要的细节
          将会造成客户端和备忘录对象持有对同一个Vector对象的引用，也可以同时修改这个Vector对象

        //负责人角色
        import java.util.Vector;

        public class Caretaker{
                private Originator o;
                private Vector mementos = new Vector();
                private int current;

                public Caretaker(Originator o){
                        this.o = o;
                        current = 0;
                }

                public int createMemento(){
                        Memento memento = o.createMemento();
                        mementos.addElement(memento);
                        return current ++;
                }
```

```
        //将发起人恢复到某个检查点
        public void restoreMemento(int index){
                Memento memento = (Memento)mementos.elementAt(index);
                o.restoreMemento(memento);
        }

        //某个检查点删除
        public void removeMemento(int index){
                mementos.removeElementAt(index);
        }
}

//客户端
public class Client{
        private static Originator o = new Originator();
        private static Caretaker c = new Caretaker(o);
        public static void main(String[] args){
                //改变状态
                o.setState("state 0");
                //建立一个检查点
                c.createMemento();
                //改变状态
                o.setState("state 1");

                c.createMemento();

                o.setState("state 2");

                c.createMemento();

                o.setState("state 3");

                c.createMemento();

                o.setState("state 4");
```

```
                        c.createMemento();

                        o.printStates();

                        //恢复到第二个检查点
                        System.out.println("Restoring to 2");

                        c.restoreMemento(2);

                        o.printStates();

                        System.out.println("Restoring to 0");

                        c.restoreMemento(0);

                        o.printStates();

                        System.out.println("Restoring to 3");

                        c.restoreMemento(3);

                        o.printStates();


                }
        }
```

自述历史模式（备忘录模式的一个变种）：

```
        //窄接口
        public interface MementoIF{}

        //发起人角色
        public class Originator{
                public String state;

                public Originator(){}
```

```
        public void changeState(String state){
                this.state = state;
                System.out.println("State has been changed to : " + state);
        }

        public Memento createMemento(){
                return new Memento(this);
        }

        public void restoreMemento(MementoIF memento){
                Memento m = (Memento)memento;
                changeState(m.state);
        }

        class Memento implements MementoIF{
                private String state;

                private String getState(){
                        return state;
                }

                private Memento(Originator o){
                        this.state = o.state;
                }
        }
}

//客户端
public class Client{
        private static Originator o;
        private static MementoIF memento;

        public static void main(String args[]){
                o = new Originator();
                o.changeState("State 1");
                memento = o.createMemento();
                o.changeState("State 2");
```

```
                o.restoreMemento(memento);
            }
        }
```

模式的优缺点：
由于“自述历史”作为一个备忘录模式的特殊实现形式非常简单易懂，它可能是备忘录模式最为流行的实现形式。

备忘录模式的操作过程
1、客户端为发起人角色创建一个备忘录对象。
2、调用发起人对象的某个操作，这个操作是可以撤销的。
3、检查发起人对象所出状态的有效性。检查的方式可以是发起人对象的内部自查，也可以由某个外部对象进行检查。
4、如果需要的话，将发起人的操作撤销，也就是说根据备忘录对象的记录，将发起人对象的状态恢复过来。

“假如”协议模式的操作过程：
1、将发起人对象做一个拷贝。
2、在拷贝上执行某个操作。
3、检查这个拷贝的状态是否有效和自恰。
4、如果检查结果是无效或者不自恰的，那么扔掉这个拷贝，并触发异常处理程序；相反，如果检查是有效和自恰的，那么在原对象上执行这个操作
显然这一做法对于撤销一个操作并恢复操作前状态较为复杂和困难的发起人对象来说是一个较为谨慎和有效的做法。

“假如”协议模式的优点和缺点
具体来说，这个做法的长处是可以保证发起人对象永远不会处于无效或不自恰的状态上，这样作的短处是成功的操作必须执行两次。
如果操作的成功率较低的话，这样做就比较划算，反之就不太划算。

使用备忘录模式的优点和缺点

一、备忘录模式的优点
1、有时一些发起人对象的内部信息必须保存在发起人对象以外的地方，但是必须要由发起人对象自己读取，这

时，

使用备忘录模式可以把复杂的发起人内部信息对其他的对象屏蔽起来，从而可以恰当地保持封装的边界。

2、本模式简化了发起人类。发起人不再需要管理和保存其内部状态的一个个版本，客户端可以自行管理他们所需

要的这些状态的版本。

3、当发起人角色的状态改变的时候，有可能这个状态无效，这时候就可以使用暂时存储起来的备忘录将状态复原。

二、备忘录模式的缺点：

1、如果发起人角色的状态需要完整地存储到备忘录对象中，那么在资源消耗上面备忘录对象会很昂贵。

2、当负责人角色将一个备忘录 存储起来的时候，负责人可能并不知道这个状态会占用多大的存储空间，从而无法

提醒用户一个操作是否很昂贵。882——P

3、当发起人角色的状态改变的时候，有可能这个协议无效。如果状态改变的成功率不高的话，不如采取“假如”协议模式。

1.18 java设计模式笔记【行为模式第二篇】

发表时间: 2009-10-23

不变（Immutable）模式
一个对象的状态在对象被创建之后就不再变化，这就是不变模式。

一、不变模式有两种模式
1、弱不变模式
一个类的实例的状态是不可变化的，但是这个类的子类的实例具有可能会变化的状态。这样的类符合弱不变模式的定义。
要实现弱不变模式，一个类必须满足下面条件：
第一、所考虑的对象没有任何方法会修改对象的状态，这样一来，当对象的构造子将对象的状态初始化之后，
对象的状态便不再改变。
第二、所有的属性都应当是私有的。不要声明任何的公共的属性，以防客户端对象直接修改任何的内部状态。
第三、这个对象所引用到的其它对象如果是可变对象的话，必须设法限制外界对这些可变对象的访问，以防止外界修改
这些对象。如果可能，应当尽量在不变对象内部初始化这些被引用到的对象，而不要在客户端初始化，然后再
传入到不变对象内部来。如果某个可变对象必须在客户端初始化，然后再传入到不变对象里的话，就应当考虑
在不变对象初始化的时候，将这个可变对象复制一份，而不再使用原来的拷贝。
弱不变模式的缺点是：
第一、一个弱不变对象的自对象可以是可变对象；换言之，一个弱不变对象的子对象可能是可变的
第二、这个可变的子对象可能可以修改父对象的状态，从而可能会允许外界修改父对象的状态。
2、强不变模式
一个类的实例的状态不会改变，同时它的子类的实例也具有不可变化的状态。这样的类符合强不变模式。
要实现强不变模式，一个类必须首先满足弱不变模式所要求的所有条件，并且还要满足下面的条件之一：
第一、所考虑的类所有的方法都应当是final：这样这个类的子类不能够置换掉此类的方法
第二、这个类本身就是final的，那么这个类就不可能会有子类，从而也就不可能有被子类修改的问题。593P

二、不变模式的优缺点
不变模式有很明显的有点：
1、因为不能修改一个不变对象的状态，所以可以避免由此引起的不必要的程序错误；换言之，一个不变模式的对象要比可变的
对象更加容易维护。
2、因为没有任何一个线程能够修改不变对象的内部状态，一个不变对象自动就是线程安全的，
这样就可以省掉处理同步化的开销。一个不变对象可以自由地被不同的客户端共享。

不变模式唯一的缺点是：
一旦需要修改一个不变对象的状态，就只好创建一个新的同类对象。在需要频繁修改不变对象的环境里，会有大量的不变对象
作为中间结果被创建出来，再被java语言的垃圾回收器收集走。这是一种资源上的浪费。

三、一个用来说明不变模式的复数类例子

```
//-----
                package day1114;

@SuppressWarnings("serial")
public final class Complex extends Number implements java.io.Serializable,
                Cloneable, Comparable {
        // 虚数单位
        public static final Complex i = new Complex(0.0, 1.0);

        // 复数的实部
        private double re;

        // 复数的虚部
        private double im;

        // 构造子，根据传进的复数再构造一个数学值的复数
        public Complex(Complex z) {
                re = z.re;
                im = z.im;
        }

        // 根据传进的实部和虚部构造一个复数对象
        public Complex(double re, double im) {
                this.re = re;
                this.im = im;
        }

        // 构造子，根据一个实部构造复数对象
        public Complex(double re) {
                this.re = re;
                this.im = 0.0;
```

```
    }

    // 默认构造子，构造一个为零的复数
    public Complex() {
            re = 0.0;
            im = 0.0;
    }

    // 把本复数与作为参数传进的复数相比较
    public boolean equals(Complex z) {
            return (re == z.re && im == z.im);
    }

    // 把本对象与作为参数传进的对象相比较
    public boolean equals(Object obj) {
            if (obj == null) {
                    return false;
            } else if (obj instanceof Complex) {
                    return equals((Complex) obj);
            } else {
                    return false;
            }
    }

    public int hashCode() {
            long re_bits = Double.doubleToLongBits(re);
            long im_bits = Double.doubleToLongBits(im);
            return (int) ((re_bits ^ im_bits) ^ ((re_bits ^ im_bits) >> 32));
    }

    // 返回本复数的实部
    public double real() {
            return re;
    }

    // 返回本复数的虚部
    public double imag() {
```

```
        return im;
    }

    // 静态方法，返还作为参数传进的复数的实部
    public static double real(Complex z) {
        return z.re;
    }

    // 静态方法，返还作为参数传进的复数的虚部
    public static double imag(Complex z) {
        return z.im;
    }

    // 静态方法，返还作为参数传进的复数的相反数
    public static Complex negate(Complex z) {
        return new Complex(-z.re, -z.im);
    }

    // 静态方法，返还作为参数传进的复数的共轭数
    public static Complex conjugate(Complex z) {
        return new Complex(z.re, -z.im);
    }

    // 静态方法，返还两个数的和
    public static Complex add(Complex x, Complex y) {
        return new Complex(x.re + y.re, x.im + y.im);
    }

    public static Complex add(Complex x, double y) {
        return new Complex(x.re + y, x.im);
    }

    public static Complex add(double x, Complex y) {
        return new Complex(x + y.re, y.im);
    }

    // 静态方法，返还两个数的差
```

```
    public static Complex subtract(Complex x, Complex y) {
            return new Complex(x.re - y.re, x.im - y.im);
    }

    public static Complex subtract(Complex x, double y) {
            return new Complex(x.re - y, x.im);
    }

    public static Complex subtract(double x, Complex y) {
            return new Complex(x - y.re, -y.im);
    }

    // 静态方法，返还两个数的乘积
    public static Complex multiply(Complex x, Complex y) {
            return new Complex(x.re * y.re - x.im * y.im, x.re * y.im + x.im * y.re);
    }

    public static Complex multiply(Complex x, double y) {
            return new Complex(x.re * y, x.im * y);
    }

    public static Complex multiply(double x, Complex y) {
            return new Complex(x * y.re, x * y.im);
    }

    public static Complex multiplyImag(Complex x, double y) {
            return new Complex(-x.im * y, x.re * y);
    }

    public static Complex multiplyImag(double x, Complex y) {
            return new Complex(-x * y.im, x * y.re);
    }

    // 静态方法，返还两个数的商
    public static Complex divide(Complex x, Complex y) {
            double a = x.re;
            double b = x.im;
```

```
        double c = y.re;
        double d = y.im;
        @SuppressWarnings("unused")
        double scale = Math.max(Math.abs(c), Math.abs(d));
        double den = c * c + d * d;
        return new Complex((a * c + b * d) / den, (b * c - a * d) / den);
    }

    public static Complex divide(Complex x, double y) {
        return new Complex(x.re / y, x.im / y);
    }

    public static Complex divide(double x, Complex y) {
        double den, t;
        Complex z;
        if (Math.abs(y.re) > Math.abs(y.im)) {
                t = y.im / y.re;
                den = y.re + y.im * t;
                z = new Complex(x / den, -x * t / den);
        } else {
                t = y.re / y.im;
                den = y.im + y.re * t;
                z = new Complex(x * t / den, -x / den);
        }
        return z;
    }

    // 静态方法，返还复数的绝对值
    public static double abs(Complex z) {
        return z.re * z.re - z.im * z.im;
    }

    // 静态方法，返还复数的相位角
    public static double argument(Complex z) {
        return Math.atan2(z.im, z.re);
    }
```

```
// 返还复数的字符串
public String toString() {
        if (im == 0.0) {
                return String.valueOf(re);
        }
        if (re == 0.0) {
                return String.valueOf(im) + "i";
        }

        String sign = ((im < 0.0) ? "" : "+");
        return (String.valueOf(re) + sign + String.valueOf(im) + "i");
}

@Override
public double doubleValue() {
        // TODO Auto-generated method stub
        return 0;
}

@Override
public float floatValue() {
        // TODO Auto-generated method stub
        return 0;
}

@Override
public int intValue() {
        // TODO Auto-generated method stub
        return 0;
}

@Override
public long longValue() {
        // TODO Auto-generated method stub
        return 0;
}
```

```
        public int compareTo(Object o) {
                // TODO Auto-generated method stub
                return 0;
        }
}

                //客户端
                public class TestComplex{
                        public static void main(String args[]){
                                Complex c1 = new Complex(10,20);
                                Complex c2 = new Complex(0,1);
                                Complex res = Complex.mnltiply(c1,c2);
                                System.out.println("Real part = " + res.real());
                                System.out.println("Imaginary part = " + res.imag());
                        }
                }
```

1.19 java设计模式笔记【行为模式第三篇】

发表时间: 2009-10-23

策略（Strategy）模式
策略模式属于对象的行为模式。其用意是针对一组算法，将每一个算法封装到具有共同接口的独立的类中，从而使得它们可以相互替换，
策略模式可以在不影响到客户端的情况下发生变化。

一、策略模式涉及到的角色;
1、环境（Context）角色：持有一个Strategy类的引用。
2、抽象策略（Strategy）角色：这是一个抽象角色，通常由一个接口或抽象类实现。此角色给出所有的具体策略类所需要的接口。
3、具体策略（ConcreteStrategy）角色：包装了相关的算法或行为。

```
//环境角色
                public class Context{
                        private Strategy strategy;

                        //策略方法
                        public void contextInterface(){
                                strategy.strategyInterface();
                        }
                }

                //抽象策略类
                public abstract class Strategy{
                        //策略方法
                        public abstract void strategyInterface();
                }

                //具体策略类
                public class ConcreteStrategy extends Strategy{
                        public void strategyInterface(){
                                //.............
                        }
                }
```

二、一个图书折扣店的例子

```
//抽象折扣类
public abstract class DiscountStrategy{
        private single price = 0;
        private int copies = 0;

        //策略方法
        public abstract single calculateDiscount();

        public DiscountStrategy(single price , int copies){
                this.price = price;
                this.copies = copies;
        }
}

//具体折扣类(没有折扣)
public class NoDiscountStrategy extends DiscountStrategy{
        private single price = 0;
        private int copies = 0;

        public  NoDiscountStrtegy(single price, int copies){
                this.price = price;
                this.copies = copies;
        }

        public single calculateDiscount(){
                return 0;
        }
}

//
public class FlatRateStrategy extends DiscountStrategy{
        private single price = 0;
        private int copies = 0;
        private single amount;
```

```
        public FlatRateStrategy(single price , int copies){
                this.price = price;
                this.copies = copies;
        }

        public single getAmount(){
                return amount;
        }

        public void setAmount(single amount){
                this.amount = amount;
        }

        public single calculateDiscount(){
                return copies*amount;
        }
}

//
public class PercentageStrategy extends DiscountStrategy{
        private single percent;
        private single price = 0;
        private int copies = 0;

        public PercentageStrategy(single price , int copies){
                this.price = price;
                this.copies = copies;
        }

        public single getPercent(){
                return percent;
        }

        public void setPercent(single percent){
                this.percent = percent;
        }
```

```
                public single calculateDiscount(){
                        return copies*price*percent;
                }
        }
```

三、使用策略模式的情况

1、如果在一个系统里面有许多类，它们之间的区别仅在于它们的行为，那么使用策略模式可以动态地让一个对象在许多行为

中选择一种行为。

2、一个系统需要动态地在几种算法中选择一种。那么这些算法可以包装到一个个的具体算法类里面，而这些具体算法类都是

一个抽象算法类的子类。换言之，这些具体算法类均有统一的接口，由于多态性原则，客户端可以选择使用任何一个具体

算法类，并只持有一个数据类型是抽象算法类的对象。

3、一个系统的算法使用的数据不可以让客户端知道。策略模式可以避免让客户大涉及到不必要接触到的复杂的和只与算法有关的数据。

4、如果一个对象有很多行为，如果不用恰当的模式，这些行为就只好使用多重的条件选择语句来实现。此时，使用策略模式，把这些

行为转移到相应的具体策略类里面，就可以避免使用难以维护的多重条件选择语句，并体现面向对象设计的概念。

四、策略模式的优点，缺点

1、优点：

（1）策略模式提供了管理相关的算法族的办法。策略类的等级结构定义了一个算法或行为族。恰当使用继承可以把

公共的代码移到父类里面，从而避免重复的代码。

（2）策略模式提供了可以替换继承关系的办法。继承可以处理多种算法或行为。如果不是用策略模式，那么使用算法

或行为的环境类就可能会有一些子类，每一个子类提供一个不同的算法或行为。但是，这样一来算法或行为的使

用者就和算法或行为本身混在一起。决定使用哪一种算法或采用哪一种行为的逻辑就和算法或行为的逻辑混合在

一起，从而不可能再独立演化。继承使得动态改变算法或行为变得不可能。

（3）使用策略模式可以避免使用多重条件转移语句。多重转移语句不易维护，它把采取哪一种算法或采取哪一种行为

的逻辑与算法或行为的逻辑混合在一起，统统列在一个多重转移语句里面，比使用继承的办法还要原始和落后。

2、缺点：

（1）客户端必须知道所有的策略类。并自行决定使用哪一个策略类。这就意味着客户端必须理解这些算法的区别，以便

适时选择恰当的算法类。换言之，策略模式只使用于客户端知道所有的算法或行为的情况。

（2）策略模式造成很多的策略类。有时候可以通过把依赖于环境的状态保存到客户端里面，而将策略类设计成可共享的，

这样策略类实例可以被不同客户端使用。换言之，可以使用享元模式来减少对象的数量。

1.20 java设计模式笔记【行为模式第四篇】

发表时间: 2009-10-23

调停者（Mediator）模式
调停者模式是对象的行为模式。调停者模式包装了一系列对象相互作用的方式，使得这些对象不必互相明显引用。从而使它们可以较松散地耦合。
当这些对象中的某些对象之间的相互作用发生改变时，不会立即影响其他的一些对象之间的相互作用。从而保证这些相互作用可以彼此独立地变化

要想恰到好处地在一个系统里面使用设计模式，必须做到以下几点：
（1）完全了解面临的问题，这就是说要完全了解具体情况。如果不完全了解所面临的问题，怎么能谈得上解决问题呢？
（2）完全了解模式，这就是说要十分懂得理论。如果不完全懂得所使用的理论，怎么能够正确地应用这一理论呢？
（3）非常了解怎样使用设计模式解决实际问题，这就是说要将模式理论与具体系统需求情况相结合。如果设计师不知道一个设计模式怎样
　对系统设计有帮助的华，最好不要使用这个模式。不要只是因为想在简历上写上设计模式方面的经验就盲目地使用模式。

调停者模式就是一个很容易被滥用的模式。

//调停者模式包括以下几种角色
1、抽象调停者角色：定义出同事对象到调停者对象的接口，其中主要的方法是一个（或者多个）事件方法，在有些情况下，
　这个抽象对象可以省略。一般而言，这个角色由一个Java抽象类或者Java对象实现。
2、具体调停者角色：从抽象调停者继承而来，实现了抽象超类所声明的事件方法。具体调停者知晓所有的具体同事类，它从具体同事
　对象接受消息、向具体同事对象发出命令。一般而言，这个角色由一个具体Java类实现。
3、抽象同事类角色：定义出调停者到同事对象的接口。同事对象只知道调停者而不知道其余的同事对象。一般而言，这个角色由一个
　Java抽象类或者Java对象实现。
4、具体同事类角色：所有的具体同时类均从抽象同事类继承而来。每一个具体同事类都很清楚它自己在小范围内的行为，而不知到它
　在大范围内的目的。

```
//抽象类同事类
                public abstract class Colleague{
```

```
        private Mediator mediator;

        public Colleague(Mediator m){
                mediator =  m;
        }

        public Mediator getMediator(){
                return mediator;
        }

        public abstract void action();

        public void change(){
                mediator.colleagueChanged(this);
        }
 }

//具体同事类
public class Colleague1 extends Colleague{
        public Colleague1(Mediator m){
                super(m);
        }

        public void action(){
                System.out.println("This is an action from Colleague 1");
        }
}

public class Colleague2 extends Colleague{
        public Colleague2(Mediator m){
                super(m);
        }

        public void action(){
                System.out.println("This is an action from Colleague 2");
        }
}
```

```
//抽象调停者角色
public class Mediator{
	public abstract void colleagueChanged(Colleague c);

	public static void main(String[] args){
		ConcreteMediator mediator = new ConcreteMediator();

		mediator.createConcreteMediator();
		Colleague c1 = new Colleague1(mediator);
		Colleague c2 = new Colleague2(mediator);
		mediator.colleagueChanged(c1);
	}
}

//具体调停者角色
public class ConcreteMediator extends Mediator{
	private Colleague1 colleague1;
	private Colleague2 colleague2;

	public void colleagueChanged(Colleague c){
		colleague1.action();
		colleague2.action();
	}

	public void createConcreteMediator(){
		colleague1 = new Colleague1(this);
		colleague2 = new Colleague2(this);
	}

	public Colleague1 getColleague1(){
		return colleague1;
	}

	public Colleague2 getColleague2(){
		return colleague2;
	}
```

```
        }
```

调停者模式的优点和缺点：

1、调停者模式的优点：

（1）适当使用调停者模式可以较少使用静态的继承关系，使得具体同事类可以更加容易地被复用。

（2）适当使用调停者模式可以避免同事之间的过渡耦合，使得调停类与同事类可以相对独立地演化。

（3）调停者模式将多对多的相互转化为一对多的相互作用，使得对象之间的关系更加易于维护个理解。

（4）调停者模式将对象的行为和协作抽象化，把对象在小尺度的行为上与其他对象的相互作用分开处理。

2、调停者模式的缺点：

（1）调停者模式降低了同事对象的复杂性，代价是增加了调停者类的复杂性。当然，在很多情况下，设置一个调停者并不比不设置一个调停者更好

（2）调停者类经常充满了各个具体同事类的关系协调代码，这种代码常常是不能复用的。因此，具体同事类的复用是以调停者类的不可复用为代价的。

显然，调停者模式为同事对象，而不是调停者对象提供了可扩展性，所以这个模式所提供的可扩展性是一种（向同事对象）倾斜的可扩展性。

在什么情况下使用调停者模式

大多数对模式的研究集中于模式应当在什么情况下使用，却往往忽视这些模式不应当在什么情况下使用。

1、不应当在责任划分混乱时使用

2、不应当对“数据类”和“方法类”使用

1.21 java设计模式笔记【行为模式第五篇】

发表时间: 2009-10-23

迭代子（Iterator）模式：
迭代子模式又叫游标（Cursor）模式，是对象的行为模式。迭代子模式可以顺序地访问一个聚集中的元素而不必暴露聚集的内部表象。

多个对象聚在一起形成的总体称之为聚集（Aggregate），聚集对象是能够包容一组对象的容器对象。聚集依赖于趋集结构的抽象化，具体复杂性
和多样性。数组就是最基本的聚集，也就是其他的java聚集对象的设计基础。
java聚集（Collection）对象是实现了共同的java.util.Collection接口的对象，是java语言对聚集概念的直接支持。

聚集对象必须提供适当的方法，允许客户端能够按照一个线性顺序遍历所有的元素对象，把元素对象提取出来或者删除掉等。一个使用聚集的系统
必然会使用这些方法操控聚集对象，因而在使用聚集的系统演化过程中，会出现两类问题：
（1）迭代逻辑没有改变，但是需要将一种聚集换成另一种聚集。因为不同的聚集具有不同的遍历接口，所以需要修改客户端代码，以便
　将已有的迭代调用换成新聚集对象所要求的接口。
（2）聚集不会改变，但是迭代方式需要改变。比如原来只需要读取元素和删除元素，但现在需要增加新的元素；或者原来的迭代仅仅遍历
　所有的元素，而现在则需要对元素加以过滤等。这时就只好修改聚集对象，修改已有的遍历方法，或者增加新的方法。

迭代子模式涉及到的几个角色：
1、抽象《迭代子》角色：此抽象角色定义出遍历元素所需的接口。
2、具体《迭代子》角色：此角色实现了Iterator接口，并保持迭代过程中的游标位置。
3、聚集角色：此抽象角色给出创建《迭代子》对象的接口。
4、具体聚集角色：实现了创建《迭代子》对象的接口，返回一个合适的具体《迭代子》实例。
5、客户端角色：持有对聚集及其《迭代子》对象的引用，调用迭代子对象的迭代接口，也有可能通过迭代子操作聚集元素的增加和删除。

宽接口和窄接口：
1、宽接口：如果一个聚集的接口提供了可以用来修改聚集元素的方法，这个接口就是所谓的宽接口
这种提供宽接口的聚集叫做白箱聚集。聚集对象向外界提供同样的宽接口。
2、窄接口：与上反之。
这种同时保证聚集对象的封装和迭代子功能的实现的方案叫做黑箱实现方案。

白箱聚集（外禀迭代子）：

```
//抽象聚集角色
public abstract class Aggregate{
	//工厂方法返回一个迭代子
	public Iterator createIterator(){
		return null;
	}
}

//抽象迭代子角色
public interface Iterator{
	//迭代方法：移动到第一个元素
	void first();

	//迭代方法：移动到下一个元素
	void next();

	//迭代方法：是否是最后一个元素
	boolean isDone();

	//迭代方法：返回当前元素
	Object currentItem();
}

//具体聚集角色
public class ConcreteAggregate extends Aggregate{
	private Object[] obj = {
		"Monk Tang",
		"Monkey",
		"Pigsy",
		"Sandy",
		"Horse"
	};

	//工厂方法：返回一个迭代子对象
	public Iterator creteIterator(){
		return new ConcreteIterator(this);
```

```
        }

        //取值方法：向外界提供聚集元素
        public Object getElement(int index){
                if(index < obj.length){
                        return obj[index];
                }else{
                        return null;
                }
        }

        //取值方法，向外界提供聚集大小
        public int size(){
                return obj.length;
        }
}
```

如果一个对象的内部状态在对象被创建之后就不再变化，这个对象就称为不变对象。如果一个聚集
那么在迭代过程中，一旦聚集元素发生改变（比如一个元素被删除，或者一个新的元素被加进来）
给出正确的结果。

```
//具体《迭代子》角色
public class ConcreteIterator implements Iterator{
        private ConcreteAggregate agg;
        private int index = 0;
        private int size = 0;

        public ConcreteIterator(ConcreteAggregate agg){
                this.agg = agg;
                size = agg.size();
                index= 0;
        }

        public void first(){
                index= 0;
        }
```

```
        public void next(){
                if(index < size){
                        index++;
                }
        }

        public boolean isDone(){
                return (index >= size);
        }

        public Object currentItem(){
                return agg.getElement(index);
        }
}

//客户端
public class Client{
        private Iterator it;
        private Aggregate agg = new ConcreteAggregate();

        public void operation(){
                it = agg.createIterator();

                while(!it.isDone()){
                        System.out.println(it.currentItem().toString());
                        it.next();
                }
        }

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

外禀迭代子的意义

客户端可以自行进行迭代，不一定非得需要一个迭代子对象。但是，迭代子对象和迭代模式会将迭代过程抽象化，将作为

迭代消费者的客户端与迭代负责人的迭代子责任分割开，使得两者可以独立演化。在聚集对象的种类发生变化，或者迭代

的方法发生改变时，迭代子作为一个中介层可以吸收变化的因素，而避免修改客户端或者聚集本身。

此外，如果系统需要同时针对几个不同的聚集对象进行迭代，而这些聚集对象所提供的遍历方法有所不同时，使用迭代子

模式和一个外界的迭代子对象是有意义的。具有同一迭代接口的不同迭代子对象处理具有不同遍历接口的聚集对象，使得

系统可以使用一个统一的迭代接口进行所有的迭代。

黑箱聚集（内禀迭代子）

一个黑箱聚集不向外部提供遍历自己元素的接口，因此，这些元素对象只可以被聚集内部成员访问。由于内禀迭代子恰好是聚集内部

的成员子类，因此，内禀迭代子对象是可以访问聚集的元素的。

```
//抽象聚集
                public abstract class Aggregate{
                        //工厂方法，返还一个迭代子对象
                        public abstract Iterator createIterator();
                }

                //抽象迭代子
                public interface Iterator{
                        //迭代方法：移动到第一个元素
                        void first();

                        void next();

                        boolean isDone();

                        Object currentItem();
                }

                //具体聚集
                public class ConcreteAggregate extends Aggregate{
                        private Object[] obj = {
```

```
                "Monk Tang",
                "Monkey",
                "Pigsy",
                "Sandy",
                "Horse"
        };

        public Iterator createIterator(){
                return new ConcreteIterator();
        }

        private class ConcreteIterator implements Iterator{
                private int currentIndex = 0;

                public void first(){
                        currentIndex = 0;
                }

                public void next(){
                        if(currentIndex < obj.length){
                                currentIndex ++;
                        }
                }

                public boolean isDone(){
                        return (currentIndex == obj.length);
                }

                public Object currentItem(){
                        return obj[currentIndex];
                }
        }
}

public class Client{
        private Iterator it;
        private Aggregate agg = new ConcreteAggregate();
```

```
                public void operation(){
                        it = agg.createIterator();
                        while(it.isDone){
                                System.out.println(it.currentItem().toString());
                                it.next();
                        }
                }

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

使用外禀迭代子和内禀迭代子的情况
一个外禀迭代子往往仅存储一个游标，因此如果有几个客户端同时进行迭代的话，那么可以使用几个外禀迭代子对象，由每一个迭代子
对象控制一个独立的游标。但是，外禀迭代子要求聚集对象向外界提供遍历方法，因此会破坏对聚集的封装。如果某一个客户端可以修
改聚集元素的话，迭代会给出不自恰的结果，甚至影响到系统其他部分的稳定性，造成系统崩溃。

使用外禀迭代子的一个重要理由是它可以被及个不同的方法和对象共同享用和控制。使用内禀迭代子的优点是它不破坏对象聚集的封装。
742P
静态迭代子和动态迭代子
静态迭代子由聚集对象创建，并持有聚集对象的一个快照，在生产后这个快照的内容就不在变化。客户端可以继续修改原聚集的内容
但是迭代子对象不会反映出聚集的新变化。
静态迭代子的好处是它的安全性和简易性，换言之，静态迭代子易于实现，不容易出现错误。但是由于静态迭代子将原聚集复制了一份
因此它的短处是对时间和内存资源的消耗。对大型的聚集对象来说。使用静态迭代子不是一个合适的选择。

动态迭代子则与静态迭代子完全相反，在迭代被产生后，迭代子保持者对聚集元素的引用，因此，任何对原聚集内容的修改都会在迭代
子对象上反映出来

FailFast的含义：如果当一个算法开始后，它的运算环境发生变化，使得算法无法进行必须的调整时，这个算法就应当立即发出故障信号。

迭代子模式的优点和缺点

1、迭代子模式的优点：

（1）、迭代子模式简化了聚集的界面。迭代子具备了一个遍历接口，这样聚集的接口就不必具备遍历接口。

（2）、每一个聚集对象都可以有一个或一个以上的迭代子对象，每一个迭代子的迭代状态可以是彼此独立的。因此，一个聚集对象可以同时有几个迭代在进行之中。

（3）、由于遍历算法被封装在迭代子角色里面，因此迭代的算法可以独立于聚集角色变化。由于客户端拿到的是一个迭代子对象，因此，不必知道聚集对象的类型，就可以读取和遍历聚集对象。这样即便聚集对象的类型发生变化，也不会影响到客户端的遍历过程。

2、迭代子模式的缺点：

（1）、迭代子模式给客户端一个聚集被顺序化的错觉，因为大多数的情况下聚集的元素并没有确定的顺序，但是迭代

必须以一定线性顺序进行。如果客户端误以为顺序是聚集本身具有的特性而过度依赖于聚集元素的顺序，很可

能得出错误的结果。

（2）、迭代子给出的聚集元素没有类型特征。一般而言，迭代子给出的元素都是Object类型，因此，客户端必须具备

这些元素类型的知识才能使用这些元素。

一个例子：（关于搜索两警察的购物）

```
//购物框抽象类
public abstract class Purchase{
        private Vector elements = new Vector(5);

        //工厂方法：提供迭代子的实例
        public abstract Iterator createIterator();

        //聚集方法：将一个新元素增加到聚集的最后面
        public void append(Object anObj){
                elements.addElement(anObj);
        }
        //聚集方法：删除一个方法
        public void remove(Object anObj){
                elements.removeElement(anObj);
```

```
        }

        public Object currentItem(int n){
                return elements.elementAt(n);
        }

        public int count(){
                return elements.size();
        }
}

//具体聚集类
public class PurchaseOfCopA extends Purchase{
        public PurchaseOfCopA(){}

        public Iterator createIterator(){
                return new ForwardIterator(this);
        }
}

public class PurchaseOfCopB extends Purchase{
        public PurchaseOfCopB(){}

        public Iterator createIterator(){
                return new BackwardIterator(this);
        }
}

//迭代子接口
public interface Iterator{
        void first();

        void next();

        boolean isDone();

        Object currentItem();
```

```
}

//具体迭代子
public class ForwardIterator implements Iterator{
        private int state;
        private Purchase obj;

        public ForwardIterator(Purchase anObj){
                obj = anObj;
        }

        public void first(){
                state = 0;
        }

        public void next(){
                if(!isDone()){
                        state++;
                }
        }

        public boolean isDone(){
                if(state > obj.count()-1){
                        return true;
                }
                return false;
        }

        public Object currentItem(){
                return obj.currentItem(state);
        }
}

public class BackwardIterator implements Iterator{
        private int state;
        private Purchase obj;
```

```
                public BackwardIterator(Purchase anObj){
                        obj = anObj;
                }

                public void first(){
                        state = obj.count()-1;
                }

                public void next(){
                        if(!isDone()){
                                state --;
                        }
                }

                public boolean isDone(){
                        if(state > 1){
                                return true;
                        }
                        return false;
                }
                public Object currentItem(){
                        return obj.currentItem(state);
                }
        }
```

1.22 java设计模式笔记【行为模式第六篇】

发表时间: 2009-10-23

访问者（Visitor）模式：
访问者模式是对象的行为模式。访问者模式的目的是封装一些施加于某种数据结构元素之上的操作。一旦这些操作要修改的话，接受这个操作的数据
结构则 可以保持不变。

聚集是的大多数的系统都要处理一种容器对象，它保存了对其他对象的引用。相信大多数读者都有处理聚集的经验，但是人家处理过的大多数聚集
恐怕 都是同类操作，而迭代子模式就是为这种情况准备的设计模式。

如果需要针对一个包含不同类型元素的聚集采取某种操作，而操作的细节根据元素的类型不同而有所不同时，就会出现必须对元素类型做类型判断
的条件转移语句。

一、访问者模式涉及到抽象访问者角色、具体访问者角色、抽象节点角色、具体节点角色、结构对象角色以及客户端角色。
1、抽象访问者角色：声明了一个或者多个访问操作，形成所有的具体元素角色必须实现的接口。
2、具体访问者角色：实现抽象访问者角色所声明的接口，也就是抽象访问者所声明的各个访问操作。
3、抽象节点角色：声明一个接受操作，接受一个访问者对象作为一个参量。
4、具体节点角色：实现了抽象元素所规定的接受操作。
5、结构对象角色：有如下的一些责任，可以遍历结构中的所有元素；如果需要，提供一个高层次的接口让访问者对象可以访问每一个元素
　如果需要，可以设计成一个复合对象或者一个聚集。

```
//抽象访问者角色
                    public interface Visitor{
                         //对于NodeA的访问操作
                         void visit(NodeA node);

                         //对于NodeB的访问操作
                         void visit(NodeB node);
                    }

                    //具体访问者角色
```

```
public class VisitorA implements Visitor{
     //对应于NodeA的访问操作
     public void visit(NodeA nodeA){
             System.out.println(nodeA.operationA());
     }

     //对应于NodeB的访问操作
     public void visit(NodeB nodeB){
             System.out.println(nodeB.operationB());
     }
}

 public class VisitorB implements Visitor{
     //对应于NodeA的访问操作
     public void visit(NodeA nodeA){
             System.out.println(nodeA.operationA());
     }

     //对应于NodeB的访问操作
     public void visit(NodeB nodeB){
             System.out.println(nodeB.operationB());
     }
}

//抽象节点角色
public abstract class Node{
     //接受操作
     public abstract void accept(Visitor visitor);
}

//模式中具体节点NodeA角色
public class NodeA extends Node{
     //接受操作
     public void accept(Visitor visitor){
             visitor.visit(this);
     }
```

```
        //NodeA特有的商业方法
        public String operationA(){
                return "NodeA is visited.";
        }
}

 public class NodeB extends Node{
        //接受操作
        public void accept(Visitor visitor){
                visitor.visit(this);
        }

        //NodeA特有的商业方法
        public String operationB(){
                return "NodeB is visited.";
        }
}

//模式中结构对象ObjectStructure
import java.util.Vector;
import java.util.Enumeration;
public class ObjectStructure{
        private Vector nodes;
        private Node node;
        public ObjectStructure(){
                nodes = new Vector();
        }

        //执行访问操作
        public void action(Visitor visitor){
                for(Enumeration e = nodes.elements(); e.hasMoreElements();){
                        node = (Node)e.nextElement();
                        node.accept(visitor);
                }
        }

        //增加一个新的元素
```

```
        public void add(Node node){
                nodes.addElements(node);
        }
}

//客户端
public class Client{
        private static ObjectStructure aObjects;
        private static Visitor visitor;

        public static void main(String args[]){
                //创建一个结构对象
                aObjects =  new ObjectStruture();

                //给结构增加一个节点
                aObjects.add(new NodeA());

                //给结构增加一个节点
                aObjects.add(new NodeB());

                //创建一个新的访问者
                visitor = new VisitorA();

                //让访问者访问结构
                aObjects.action(visitor);
        }
}
```

二、访问者模式仅应当在被访问的类结构非常稳定的情况下使用。换言之，系统很少出现需要加入新节点的情况。如果出现需要加入的新节点

这时就必须在每一个访问对象里加入一个对应于这个新节点的访问操作，而这是对一个系统的大规模修改，因而是违背“开闭”原则的。

访问这模式允许在节点中加入新的方法，相应的仅仅需要在一个新的访问者类中加入此方法，而不需要在每一个访问者类中都加入此方法。

显然，访问者模式提供了倾斜的可扩展性设计：方法集合的可扩展性和类集合的不可扩展性。

换言之，如果系统的数据结构是颇繁变化的，则不适合使用访问者模式。

三、使用访问者模式的优缺点：

1、优点：

（1）访问者模式使得增加新的操作变得很容易。如果一些操作依赖于一个复杂的结构对象的话，那么一般而言，增加

新的操作会很复杂。而使用访问者模式，增加新的操作已意味这增加一个新的访问者类，因此，变得很容易。

（2）访问者模式将有关的行为几种到一个访问者对象中，而不是分散到一个个的节点类中。

（3）访问者模式可以跨过几个类的等级结构访问属于不同的等级结构的成员类。迭代子只能访问属于同一个类型等级结构

的成员对象，而不能访问属于不同等级结构的对象。访问者模式可以做到这一点。

（4）积累状态。每一个单独的访问者对象都集中了相关的行为，从而也就可以在访问的过程中将执行操作的状态积累在自己

内部，而不是分散到很多的节点对象中。这是有益于系统维护的优点。

2、缺点：

（1）增加新的节点类变得很困难。没增加一个新的节点都意味着要在抽象访问者角色中增加一个新的抽象操作，并在每一个

具体访问者类中增加相应的具体操作。

（2）破坏封装。访问者模式要求访问者对象并调用每一个节点对象的操作，这隐含了一个对所有节点对象的要求：它们必须

暴露一些自己的操作和内部状态，从而使这些状态不再存储在节点对象中，这也是破坏封装的。

当实现访问者模式时，要将尽可能多的对象浏览逻辑放在（抽象访问者）Visitor类中，而不是放在它的子类里。

这样的话，具体访问者类对所有的对象结构依赖较少，从而使维护较为容易。

负责遍历行为的，可供选择的有：结构对象、访问者对象或者创建一个新的迭代对象。

（1）由结构对象负责迭代是最常见的选择，这也就是需要结构对象角色的出发点。

（2）另一个解决方案是使用一个迭代对象来负责遍历行为。

（3）将遍历行为放到访问者中是第三个可能的选择。一般而言，这样做会导致每一个具体访问者都不得不具有

管理聚集

的内部功能，而这在一般情况下是不理想的。然而如果遍历的逻辑较为复杂的话，将所有的遍历逻辑方到结构对象

角色中，不如这种做法也不失为一种可行的选择。

```
//一个电脑专卖系统
                //抽象访问者角色
                public abstract class Visitor{
                        public abstract void visitHardDisk(HardDisk e);
                        public abstract void visitMainBoard(MainBoard e);
                        public abstract void visitCpu(Cpu e);
                        public abstract void visitPc(Pc e);
                        public abstract void visitCase(Case e);
                        public abstract void visitIntegratedBoard(IntegratedBoard e);
                }

                //具体访问者角色
                public class PriceVisitor extends Visitor{
                        private float total;

                        public PriceVisitor(){
                                total = 0;
                        }

                        public float value(){
                                return total;
                        }

                        public void visitHardDisk(HardDisk e){
                                total += e.price();
                        }

                        public void visitMainBoard(MainBoard e){
                                total += e.price();
                        }

                        public void visitCpu(Cpu e){
```

```
                total += e.price();
        }

        public void visitPc(Pc e){
                total += e,price();
        }

        public void visitCase(Case e){
                total += e.price();
        }

        public void visitIntegratedBoard(IntegratedBoard e){
                total += e.price();
        }
}

//具体访问角色
import java.util.Vector;
public class InventoryVisitor extends Visitor{
        private Vector inv;

        public InventoryVisitor(){
                inv = new Vector(10.5);
        }

        public int size(){
                return inv.size();
        }

        public void visitHardDisk(HardDisk e){
                inv.addElement(e);
        }

        public void visitMainBoard(MainBoard e){
                inv.addElement(e);
        }
```

```
        public void visitCpu(Cpu e){
                inv.addElement(e);
        }

        public void visitPc(Pc e){
                inv.addElement(e);
        }

        public void visitCase(Case e){
                inv.addElement(e);
        }

        public void visitIntegratedBoard(IntegratedBoard e){
                inv.addElement(e);
        }
}

//抽象节点角色
public abstract class Equipment{
        //接受方法
        public abstract void accept(Visitor vis);

        //商业方法
        public abstract double price();
}

//具体节点角色
public class MainBoard extends Equipment{
        public double price(){
                return 100.00;
        }

        public void accept(Visitor v){
                System.out.println("MainBoard has been visited.");
                v.visitMainBoard(this);
        }
}
```

```
//具体节点角色
public class HardDisk extends Equipment{
        public double price(){
                return 200.00;
        }

        public void accept(Visitor v){
                System.out.println("HardDisk has been visited.");
                v.visitHardDisk(this);
        }
}

//具体节点角色
public class Cpu extends Equipment{
        public double price(){
                return 800.00;
        }

        public void accept(Visitor v){
                System.out.println("CPU has been visited.");
                v.visitCpu(this);
        }
}

//具体节点角色
public class Case extends Equipment{
        public double price(){
                return 30.00;
        }

        public void accept(Visitor v){
                System.out.println("Case has been visited.");
                v.visitCase(this);
        }
}
```

```
//抽象复合节点
public abstract class Composite extends Equipment{
        private Vector parts = new Vector(10);

        public Composite(){}

        public double price(){
                double total = 0;
                for(int i = 0; i < parts.size(); i ++){
                        total += ((Equipment)parts.get(i)).price();
                }
                return total;
        }

        public void accept(Visitor v){
                for(int i = 0; i < parts.size(); i ++){
                        ((Equipment)parts.get(i)).accept(v);
                }
        }
}

 //具体复合节点角色
 public class IntegratedBoard extends Composite{
        public IntegratedBoard(){
                super.add(new MainBoard());
                super.add(new Cpu());
        }

        public void accept(Visitor v){
                System.out.println("IntegratedBoard has been visited.");
                super.accept(v);
        }
 }

  //具体复合节点角色
 public class Pc extends Composite{
        public Pc(){
```

```
                super.add(new IntegratedBoard());
                super.add(new HardDisk());
                super.add(new Case());
        }

        public void accept(Visitor v){
                System.out.println("Pc has been visited.");
                super.accept(v);
        }
 }


//客户端
public class Client{
        private static PriceVisitor pv;
        private static InventoryVisitor iv;
        private static Equipment equip;

        public static void main(String args[]){
                equip = new Pc();
                pv = new PriceVisitor();
                equip.accept(pv);
                System.out.println("Price: " + pv.value());
                System.out.println();

                iv = new InventoryVisitor();
                equip.accept(iv);
                System.out.println("Number of parts: " + iv.size());
        }
}
```

1.23 java设计模式笔记【行为模式第七篇】

发表时间: 2009-10-23

观察者（Observer）模式
观察者模式是对象的行为模式，又叫做发布 - 订阅（Publish / Subscribe）模式、模型 - 视图（Model / View）模式、
源 - 监听器（Source / Listener）模式或从属者（Dependents）模式。

观察者模式定义了一种一对多的依赖关系，许多个观察者对象同时监听某一个主题对象。这个主题对象在状态上发生变化时，
会通知所有观察者对象，使它们能够自动更新自己。657P

一、观察者模式有以下几种结构：
1、抽象主题（Subject）角色：主题角色把所有对观察者对象的引用保存在一个聚集（比如Vector对象）里，每个主题都可以有任何
数量的观察者。抽象主题提供一个接口，可以增加和删除观察者对象，主题角色又叫做抽象被观察者（Observable）角色。

2、抽象观察者（Observer）角色：为所有的具体观察者定义一个接口，在得到主题的通知时更新自己。这个接口叫做更新接口。抽象
观察者角色一般用一个抽象观察者角色一般用一个抽象或者一个接口实现，其中的方法叫做更新方法。

3、具体主题（ConcreteSubject）角色：将有关状态存入具体观察者对象；在具体主题的内部状态改变时，给所有登记过的观察者发出
通知。具体主题角色又叫做具体被观察者角色（Concrete Observerable）。具体主题角色通常用一个具体子类实现，具体主题角色
负责实现对观察者引用的聚集的管理方法。

4、具体观察者角色：存储与主题的状态自恰的状态。具体观察者角色实现抽象观察者角色所要求的更新接口，以便使本身的状态与
主题的状态相协调。如果需要，具体观察者角色可以保存一个指向具体主题对象的引用。具体观察者角色通常用一个具体子类实现。

第一种实现：由具体主题角色关联到抽象观察者角色

```
//抽象主题
public interface Subject{
        //d调用这个方法登记一个新的观察者对象
```

```
        public void attach(Observer observer);

        //调用这个方法删除一个已经登记过的观察者对象
        public void detach(Observer observer);

        //调用这个方法通知所有登记过的观察者对象
        void notifyObservers();
}

//具体主题角色
import java.util.Vector;
import java.util.Enumeration;

public class ConcreteSubject implements Subject{
        private Vector observersVector = new Vector();

        public void attach(Observer observer){
                observersVector.addElement(observer);
        }

        public void detach(Observer observer){
                observersVector.removeElement(observer);
        }

        public void notifyObservers(){
                Enumeration enumeration = observers();

                while(enumeration.hasMoreElements()){
                        ((Observer)enumeration.nextElement()).update();
                }
        }

        //这个方法给出观察者聚集的Enumeration对象
        public Enumeration observers(){
                return ((Vector)observersVector.clone()).elements();
        }
}
```

```
//抽象观察者
public interface Observer{
        void update();
}

public class ConcreteObserver implements Observer{
        public void update(){
                System.out.println("I am notified");
        }
}

//第二种实现：由抽象主题角色代表出观察者对象的聚集连线到抽象观察者
//抽象主题
import java.util.Vector;
import java.util.Enumeration;

public abstract class Subject{
        private Vector observersVector = new Vector();

        public void attach(Observer observer){
                observersVector.addElement(observer);
                System.out.println("Attach an observer.");
        }

        public void detach(Observer observer){
                observersVector.removeElement(observer);
        }

        public void notifyObservers(){
                Enumeration enumeration = observers();
                while(enumeration.hasMoreElement()){
                        ((Observer)enumeration.nextElement()).update();
                }
        }

        public Enumeration observers(){
```

```
                return ((Vector)observersVector.clone()).elements();
        }
}

//具体主题
public class ConcreteSubject extends Subject{
        private String state;

        public void change(String newState){
                state = newState;
                this.notifyObservers();
        }
}
/.................

//客户端
public class Client{
        private static ConcreteSubject subject;
        private static Observer observer;

        public static void main(String args[]){
                //创建主题对象
                subject = new ConcreteSubject();
                //创建观察者对象
                observer = new ConcreteObserver();

                subject.attach(observer);

                subject.change("new state");
        }
}
```

二、观察者模式的优点和缺点

（1）优点

1、观察者模式在被观察者和观察者之间建立一个抽象的耦合，被观察者的接口。被观察者并不认识任何一个具

体观察者，

它只知道它们都有一个共同的接口。由于被观察者个观察者没有紧密地耦合在一起，因此它们可以属于不同的抽象化层次。如果被观察者和观察者都扔到一起，那么这个对象必然跨越抽象化和具体化层次。

2、观察者模式支持广播通信。被观察者会所有的登记过的观察者发出通知。

（2）缺点

1、如果一个被观察者对象有许多直接和间接的观察者的话，将所有的观察者都通知到会花费很多时间。

2、如果在被观察者之间有循环依赖的话，被观察者会触发它们之间进行循环调用，导致系统崩溃。在使用观察者模式时

要特别注意这一点。

3、如果对观察者的通知是通过另外的线程进行异步投递的话，系统必须保证投递是以自恰的方式进行的。

4、虽然观察者模式可以随时使观察者知道所观察的对象发生了变化，但是观察者模式没有相应的机制使观察者知道所观察

的对象是怎么发生变化的。

1.24 java设计模式笔记【行为模式第八篇】

发表时间: 2009-10-23

解释器（Interpreter）模式：
解释器模式是类的行为模式。给定一个语言之后，解释器模式可以定义出其文法的一种表示，并同时提供一个解释器。客户端可以使用这个解释器
来解释这个语言中的句子。

一、解释器模式所涉及的角色
1、抽象表达式角色：声明一个所有的具体表达式角色都需要实现的抽象接口。这个接口主要是一个interpret()方法，称做解释操作。
2、终结符表达式角色：这是一个具体角色。
（1）实现了抽象表达式角色所要求的接口，主要是一个interpret()方法；
（2）文法中的每一个终结符都有一个具体终结表达式与之相对应。
3、非终结符表达式角色：这是一个具体角色。
（1）文法中的每一条规则 R=R1R2.....Rn 都需要一个具体的非终结符表达式类；
（2）对每一个 R1R2.....Rn 中的符号都持有一个静态类型为Expression的实例变量。
（3）实现解释操作，即 interpret()方法。解释操作以递归方式调用上面所提到的代表 R1R2.....Rn 中的各个符号的实
例变量
4、客户端角色：代表模式的客户端它有以下功能
（1）建造一个抽象语法树（AST或者Abstract Syntax Tree）
（2）调用解释操作interpret()。
5、环境角色：（在一般情况下，模式还需要一个环境角色）提供解释器之外的一些全局信息，比如变量的真实量值等。

（抽象语法树的每一个节点都代表一个语句，而在每一个节点上都可以执行解释方法。这个解释方法的执行就代表这个语句被解释。
由于每一个语句都代表对一个问题实例的解答。）

```
//抽象角色Expression
                /*
                      这个抽象类代表终结类和非终结类的抽象化
                      其中终结类和非终结类来自下面的文法
                      Expression ::=
                                        Expression AND Expression
```

```
                        | Expression OR Expression
                        | NOT Expression
                        | Variable
                        | Constant
        Variable ::= ....//可以打印出的非空白字符串
        Constant ::= "true" | "false"
*/

public abstract class Expression{
        //以环境类为准，本方法解释给定的任何一个表达式
        public abstract boolean interpret(Context ctx);

        //检验两个表达式在结构上是否相同
        public abstract boolean equals(Object o);

        //返回表达式的hash code
        public abstract int hashCode();

        //将表达式转换成字符串
        public abstract String toString();
}

public class Constant extends Expression{
        private boolean value;

        public Constant(boolean value){
                this.value = value;
        }
        //解释操作
        public boolean interpret(Context ctx){
                return value;
        }

        //检验两个表达式在结构上是否相同
        public boolean equals(Object o){
                if(o != null && o instanceof Constant){
                        return this.value == ((Constant)o).value;
```

```
                }
                return false;
        }

        //返回表达式的hash code
        public int hashCode(){
                return (this.toString()).hashCode();
        }

        //将表达式转换成字符串
        public String toString(){
                return new Boolean(value).toString();
        }
}

public class Variable extends Expression{
        private String name;
        public Variable(String name){
                this.name = name;
        }

        public boolean interpret(Context ctx){
                return ctx.lookup(this);
        }

        public boolean equals(Object o){
                if(o != null && o instanceof Variable){
                        return this.name.equals(((Variable)o).name);
                }
                return false;
        }

        public int hashCode(){
                return (this.toString()).hashCode();
        }

        public String toString(){
```

```
                return name;
        }
}

public class And extends Expression{
        private Expression left,right;

        public And(Expression left,Expression right){
                this.left = left;
                this.right = right;
        }

        public boolean interpret(Context ctx){
                return left.interpret(ctx) &&
                        right.interpret(ctx);
        }

        public boolean equlas(Object o){
                if(o != null && o instanceof And){
                        return this.left.equals(((And)o).left) &&
                                this.right.equals(((And)o).right);
                }
                return false;
        }

        public int hashCode(){
                return (this.toString()).hashCode();
        }

        public String toString(){
                return "(" + left.toString() + "AND" + right.toString() + ")";
        }
}

public class Or extends Expression{
        private Expression left , right;
```

```
        public Or(Expression left,Expression right){
                this.left = left;
                this.right = right;
        }

        public boolean interpret(Context ctx){
                return left.interpret(ctx) || right.interpret(ctx);
        }

        public boolean equals(Object o){
                if(o != null && o instanceof Or){
                        return this.left.equals(((And)o).left) &&
                                this.right.equals(((And)o).right);
                }
                return false;
        }

        public int hashCode(){
                return (this.toString()).hashCode();
        }

        public String toString(){
                return "(" + left.toString() + "OR" + right.toString() + ")";
        }
}

 public class Not extends Expression{
        private Expression exp;

        public Not(Expression exp){
                this.exp = exp;
        }

        public boolean interpret(Context ctx){
                return !exp.interpret(ctx);
        }
```

```
        public boolean equals(Object o){
                if(o != null && o instanceof Not){
                        return this.exp.equals(((Not)o).exp);
                }
                return false;
        }

        public int hashCode(){
                return (this.toString()).hashCode();
        }

        public String toString(){
                return "(NOT" + exp.toString() + ")";
        }
}

import java.util.HashMap;

public class Context{
        private HashMap map = new HashMap();
        public void assign(Variable var,boolean value){
                map.put(var,new Boolean(value));
        }

        public boolean lookup(Variable var) throws IllegalArgumentException{
                Boolean value = (Boolean)map.get(var);
                if(value == null){
                        throw new IllegalArgumentException();
                }
                return value.booleanValue();
        }
}

//客户端
public class Client{
        private static Context ctx;
        private static Expression exp;
```

```
        public static void main(String args[]){
                ctx = new Context();
                Variable x = new Variable("x");
                Variable y = new Variable("y");

                Constant c = new Constant(true);
                ctx.assign(x,false);
                ctx.assign(y,true);

                exp = new Or(new And(c,x),new And(y,new Not(x)));
                System.out.println("x = " + x.interpret(ctx));
                System.out.println("y = " + y.interpret(ctx));
                System.out.println(exp.toString() + "=" + exp.interpret(ctx));
        }
}
```

二、解释器模式适用于以下的情况：

（1）系统有一个简单的语言可供解释

（2）一些重复发生的问题可以用这种简单的语言表达

（3）效率不是主要的考虑。

1.25 java设计模式笔记【行为模式第九篇】

发表时间: 2009-10-23

命令（Command）模式：
命令模式属于对象的行为模式。命令模式又称为行动模式或交易模式。
命令模式把一个请求或者操作封装到一个对象中。命令模式允许系统使用不同的请求把客户端参数化，对请求排队或者记录请求日志，可以提供命令
的撤销和恢复功能。

命令模式是对命令的封装。命令模式把发出命令的责任和执行命令的责任分割开，委派给不同的对象。每一个命令都是一个操作：
请求的一方发出请求要求执行一个操作；接受的一方收到请求，并执行操作。命令模式允许请求的一方和接收的一方独立开来，使得请求的一方不必
知道接收请求的一方的接口，更不必知道请求是怎么被接收，以及操作是否被执行、何时被执行，以及是怎么被执行的。

命令允许请求的一方和接收请求的一方能够独立演化，从而具有一下的优点：
（1）命令模式使新的命令很容易地被加入到系统里。
（2）允许接收请求的一方决定是否要否决要求。
（3）能较容易地设计一个命令队列。
（4）可以容易地实现对请求的Undo和Redo。
（5）在需要的情况下，可以较容易地将命令记录日志。

一、命令模式涉及到的五个角色：
1、客户角色：创建了一个具体命令对象并确定其接收者。
2、命令角色：声明了一个给所有具体命令类的抽象接口。这是一个抽象角色，通常由一个java接口或java抽象类实现。
3、具体命令角色：定义一个接受者和行为之间的弱耦合；实现execute方法，负责调用接收者的相应操作。execute方法通常叫做
执行方法。
4、请求者（Invoke）角色：负责调用命令对象执行请求，相关的方法叫做行动方法。
5、接收者（Receiver）角色：负责具体实施和执行一个请求。任何一个类都可以成为接收者，实施和执行请求的方法叫做行动方法。

```
//客户端
                public class Client{
                        public static void main(String[] args){
```

```
                Receiver receiver = new Receiver();
                Command command = new ConcreteCommand(receiver);
                Invoker invoker = new Invoker(command);
                invoker.action();
        }
}

//请求者角色
public class Invoker{
        private Command command;

        public Invoker(Command command){
                this.command = command;
        }

        public void action(){
                command.execute();
        }
}

//接收者
public class Receiver{

        public Receiver(){
                //.............
        }

        //行动方法
        public void action(){
                System.out.println("Action has been taken.");
        }
}

//抽象命令角色
public interface Command{
        void execute();
}
```

```
        //具体命令类
        public class ConcreteCommand implements Command{
                private Receiver receiver;

                public ConcreteCommand(Receiver receiver){
                        this.receiver = receiver;
                }

                public void execute(){
                        receiver.action();
                }
        }
```

命令模式的活动序列

（1）客户端创建了一个ConcreteCommand对象，并指明了接收者；

（2）请求者对象保存了ConcreteCommand对象；

（3）请求者对象通过调用action()方法发出请求。如果命令是能撤销（Undo）的，那么调用execute()方法之前的状态。

（4）ConcreteCommand对象调用接收的一方的方法执行请求。

```
//一个例子（创世纪）
//抽象命令类
public interface CommandFromGod{
        public void execute();
}

//请求角色
import java.awt.*;
import java.awt.event.*;

public class TheWorld extends Frame implements ActionListener{
        private LetThereBeLightCommand btnLight;
        private LetThereBeLandCommand btnLand;
        private ResetCommand btnReset;
        private GodRestsCommand btnExit;
        private Panel p;
```

```
        public TheWorld(){
                super("This is the world,and God says...");
                p = new Panel();
                p.setBackground(Color.black);

                add(p);

                btnLight = new LetThereBeLightCommand("Let there be light",p);
                btnLand = new LetThereBeLandCommand("Let there be land",p);
                btnReset = new ResetCommand("Reset",p);
                btnExit = new GodRestsCommand("God rests");

                p.add(btnLight);
                p.add(btnLand);
                p.add(btnReset);
                p.add(btnExit);

                btnLight.addActionListener(this);
                btnLand.addActionListener(this);
                btnReset.addActionListener(this);
                btnExit.addActionListener(this);
                setBounds(100,100,400,200);
                setVisible(true);
        }

        public void actionPerformed(ActionEvent e){
                Command obj = (Command)e.getSource();
                obj.execute();
        }

        public static void main(String[] args){
                new TheWorld();
        }
}

import java.awt.*;
```

```
import java.awt.event.*;

public class LetThereBeLightCommand extends Button implements CommandFromGod{
        private Panel p;

        public LetThereBeLightCommand(String caption,Panel p){
                super(caption);
                this.p = p;
        }

        public void execute(){
                p.setBackground(Color.white);
        }
}

import java.awt.*;
import java.awt.event.*;

public class LetThereBeLandCommand extends Button implements CommandFromGod{
        private Panel p;

        public LetThereBeLandCommand(String caption,Panel p){
                super(caption);
                this.p = p;
        }

        public void execute(){
                p.setBackground(Color.orange);
        }
}

import java.awt.*;
import java.awt.event.*;

public class ResetCommand extends Button implements CommandFromGod{
        private Panel p;
```

```
        public ResetCommand(String caption,Panel p){
                super(caption);
                this.p = p;
        }

        public void execute(){
                p.setBackground(Color.black);
        }
}

import java.awt.*;
import java.awt.event.*;

public class GodRestsCommand extends Button implements CommandFromGod{
        public GodRestsCommand(String caption){
                super(caption);
        }

        public void execute(){
                System.exit(0);
        }
}
```

二、在什么情况下应当使用命令模式

1、使用命令模式作为“回呼”在面向对象系统中的替代。“回呼”讲的便是先将一个函数登记上，然后在以后调用此函数。

2、需要在不同的时间指定请求，将请求排队。一个命令对象和原先的请求发出者可以有不同的生命期。换言之，原先的请求

发出者可能已经不在了，而命令对象本身仍然是活动的。这时命令的接收者可以是在本地，也可以在网络的另外一个地址。

命令对象可以在串行化之后传送到另外一台机器上去。

3、系统需要支持命令的撤销（Undo）。命令对象可以把状态存储起来，等到客户端需要撤销命令所产生的效果时，可以调用

undo（）方法，把命令所产生的效果撤销掉。命令对象还可以提供redo（）方法，以供客户端在需要时，再重新实施命令

的效果。

4、如果一个系统要将系统中所有的数据更新到日志里，以便在系统崩溃时，可以根据日志里读回所有的数据更新命令，重新调用

execute()方法一条一条执行这些命令，从而恢复系统在崩溃前所做的数据更新。

5、一个系统需要支持交易（transaction）。一个交易结构封装了一组数据更新命令。使用命令模式来实现交易结构可以使系统

增加新的交易类型。826P；

三、使用命令模式的优点和缺点

1、优点

（1）命令模式把请求一个操作的对象与知道怎么执行一个操作的对象分割开。

（2）命令类与其他任何别的类一样，可以修改和推广。

（3）你可以把命令对象聚合在一起，合成为合成命令。比如上面的例子里所讨论的宏命令便是合成命令的例子。合成命令

是合成模式的应用。

（4）由于加进新的具体命令类不影响其他的类，因此增加新的具体命令类很容易。

2、缺点

使用命令模式会导致某些系统有过多的具体命令类。某些系统可能需要几十个，几百个甚至几千个具体命令类，这会使命令模式

在这样的系统里变得不实际。

1.26 java设计模式笔记【行为模式第十篇】

发表时间: 2009-10-23

模板方法（Template Method）模式
模板方法模式是类的行为模式。准备一个抽象类，将部分逻辑以具体方法以及具体构造子的形式实现，然后声明一些抽象方法来迫使子类
实现剩余的逻辑。不同的子类可以以不同的方式实现这些抽象方法，从而对剩余的逻辑有不同的实现。这就是模板方法模式的用意

一、涉及到的角色
抽象模板角色有如下责任：
1、定义了一个或多个抽象操作，以便让子类实现。这些抽象操作叫做基本操作，它们是一个顶极逻辑的组成步骤。
2、定义并实现了一个模板方法。这个模板方法一般是一个具体方法，它给出了一个顶极逻辑的骨架，而逻辑的组成
步骤在相应的抽象操作中，推迟到子类实现。顶极逻辑也有可能调用一些具体方法。

具体模板角色有如下责任：
1、实现父类所定义的一个或多个抽象方法，它们是一个顶极逻辑的组成步骤。
2、每一个抽象模板角色都可以有任意多个具体模板角色与之对应，而每一个具体模板角色可以给出这些抽象方法（也就
是顶极逻辑的组成步骤）的不同实现，从而使得顶极逻辑的实现各不相同。

```
//抽象模板类
                public abstract class AbstractClass{
                        //模板方法的声明和实现
                        public void TemplateMethod(){
                                //调用基本方法（由子类实现，以下都是）
                                doOperation1();

                                doOperation2();

                                doOperation3();(已经实现)
                        }

                        protected abstract void doOperation1();
```

```
        protected abstract void doOperation2();

        protected abstract void doOperation3(){
                //..............
        }(已经实现)
}

//具体模板类
public class ConcreteClass extends AbstractClass{
        public void doOperation1(){
                System.out.println("doOperation1()");
        }

        public void doOperation2(){
                //doOperation3();像这样的调用不要发生
                System.out.println("doOperation2()");
        }
}
```

二、一个帐户存储的例子

```
//抽象类
public abstratc class Account{
        protected String accountNumber;

        public Account(){
                accountNumber = null;
        }

        public Account(String accountNumber){
                this.accountNumber = accountNumber;
        }

        //模板方法，计算利息数额
        public final double calculateInterest(){
                double interestRate = doCalculateInterestRate();
                String accountType = doCalculateAccountType();
```

```
                double amount = calculateAmount(accountType,accountNumber);

                return amount*interestRate;
        }

        //基本方法留给子类实现
        protected abstract String doCalculateAccountType();

        protected abstract double doCalculateInterestRate();

        //基本方法，已经实现
        public final double calculateAmount(String accountType,double accountNu
                //从数据库中取数据
                return 7243.00D;
        }
}

//具体子类
public class MoneyMarketAccount extends Account{
        public String doCalculateAccountType(){
                return "Money Market";
        }

        public double doCalculateInterestRate(){
                return 0.045D;
        }
}

public class CDAccount extends Account{
        public String doCalculateAccountType(){
                return "Certificate of Deposite";
        }

        public double doCalculateInterestRate(){
                return 0.065D;
        }
}
```

```
//客户端
public class Client{
        private static Account acct = null;

        public static void main(String args[]){
                acct = new MoneyMarketAccount();
                System.out.println("Interest from Money Market account" + acct.
                acct = new CDAccount();
                System.out.println("Interest from CD account" + acct.calculatel
        }
}
```

1.27 java设计模式笔记【行为模式第十一篇】

发表时间: 2009-10-23

责任链（Chain of Responsibility）模式
责任链模式是一种对象行为模式。
在责任链模式里，很多对象由每一个对象对其下家的引用而连接起来形成一条链。请求在这个链上传递，直到链上的某一个对象决定处理此请求。
发出这个请求的客户端并不知道链上的哪一个对象最终处理这个请求，这使得系统可以在不影响客户端的情况下动态地重新组织链和分配责任。

责任链模式的角色：
1、抽象处理者角色：定义出一个处理请求的接口。如果需要，接口可以定义出一个方法，以设定和返回对下家的引用。这个角色通常由
一个java抽象类或者java接口实现。

2、具体处理者角色：具体处理者接到请求后，可以选择将请求处理掉，或者将请求传给下家，由于具体处理者持有对下家的引用，因此
，如果需要，具体处理者可以访问下家。

```
//抽象处理者
                public abstract class Hander{
                        //处理方法，调用此方法处理请求
                        protected Handle successor;
                        public abstract void handleRequest();

                        //调用此方法，设定下家
                        public void setSuccessor(Handler successor){
                                this.successor = successor;
                        }

                        public Handler getSuccessor(){
                                return successor;
                        }
                }

                //具体处理者
                public class ConcreteHandler extends Handler{
```

```
                public void handlerRequest(){
                        if(getSuccessor() != null){
                                System.out.println("The request is passed to " + getSuc
                                getSuccessor().handlerRequest();
                        }else{
                                System.out.println("The request is handled here.");
                        }
                }
        }

        //客户端
        public class Client{
                private static Handler handler1, handler2;

                public static void main(String args[]){
                        handler1 = new ConcreteHandler();
                        handler2 = new ConcreteHandler();
                        handler1.setSuccessor(handler2);
                        handler1.handleRequest();
                }
        }
```

纯的与不纯的责任链模式

一个纯的责任链模式要求一个具体的处理者对象只能在两个行为中选择一个：一是承担责任，二是把责任推给下家。

不允许出现某一个具体处理者对象在承担了一部分责任后又把责任向下传的情况。

在一个纯的责任链模式里面，一个请求必须被某一个处理者对象所接受，在一个不纯的责任链模式里面，一个请求可以最终不被

任何接受端对象所接收。

```
//一个贾家击鼓传花喝酒的例子
        //击鼓者(客户端)
        public class DrumBeater{
```

```
        private static Player player;

        public static void main(String[] args){
                //创建责任链
                player  = new JiaMu(new JiaShe(new JiaZheng(new JiaBaoYu(new Ji

                //规定由第四个处理者处理请求
                player.handler(4);
        }
}
```

抽象传花者

```
public abstract class Player{
        public abstract void handler(int i);

        private Player successor;

        public Player(){
                successor = null;
        }

        protected void setSuccessor(Player aSuccessor){
                successor = aSuccessor;
        }

        //将花传给下家，如果没有下家，系统停止运行
        public void next(int index){
                //判断下家对象是否有效
                if(successor != null){
                        //将请求传给下家
                        successor.handler(index);
                }else{
                        //系统停止运行
                        System.out.println("Program terminated.");
                        System.exit(0);
                }
        }
```

```
	}

	//JiaMu类
	public class JiaMu extends Player{
		public JiaMu(Player aSuccessor){
			this.setSuccessor(aSuccessor);
		}

		public void handler(int i){
			if(i == 1){
				System.out.println("Jia Mu gotta drink!");
			}else{
				System.out.println("JiaMu passed!");
				next(i);
			}
		}
	}

	//JiaShe

	public class JiaShe extends Player{
		public JiaShe(Player aSuccessor){
			this.setSuccessor(aSuccessor);
		}

		public void handler(int i){
			if(i == 2){
				System.out.println("Jia She gotta drink!");
			}else{
				System.out.println("Jia She passed!");
				next(i);
			}
		}
	}

	public class JiaZheng extends Player{
		public JiaZheng(Player aSuccessor){
```

```
                this.setSuccessor(aSuccessor);
        }

        public void handler(int i){
                if(i == 3){
                        System.out.println("Jia Zheng gotta drink!");
                }else{
                        System.out.println("Jia Zheng passed!");
                        next(i);
                }
        }
}

public class JiaBaoYu extends Player{
        public JiaBaoYu(Player aSuccessor){
                this.setSuccessor(aSuccessor);
        }

        public void handler(int i){
                if(i == 4){
                        System.out.println("Jia Bao Yu gotta drink!");
                }else{
                        System.out.println("Jia Bao Yu passed!");
                        next(i);
                }
        }
}

public class JiaHuan extends Player{
        public JiaHuan(Player aSuccessor){
                this.setSuccessor(aSuccessor);
        }

        public void handler(int i){
                if(i == 5){
                        System.out.println("Jia Huan gotta drink!");
                }else{
```

```
                                System.out.println("Jia Huan passed!");
                                next(i);
                        }
                }
        }
```

在下面的情况下可以使用责任链模式：
（1）系统已经有一个由处理者对象组成的链。这个链可能由合成模式给出。
（2）有多于一个的处理者对象会处理一个请求，而且事先并不知道到底由哪一处理者对象处理一个请求。
这个处理者对象是动态确定的。
（3）系统想发出一个请求给多个处理者对象中的某一个，但是不明显指定是哪一个处理者对象会处理此请求。
（4）处理一个请求的处理者对象的集合需要动态地指定时。
责任链模式降低了发出命令的对象和处理命令的对象之间的耦合，它允许多于一个的处理者对象根据自己的逻辑来决定
哪一个处理者最终处理这个命令。换言之，发出命令的对象只是把命令传给链结构的起始者，而不需要知道到底是链上
的哪一个节点处理了这个命令。

显然。这意味着在处理命令时，允许系统有更多的灵活性。哪一个对象最终处理一个命令可以因为由哪些对象参加责任链，
以及这些对象在责任链上的位置不同而有所不同。

```
//一个使用Timer的例子
        import java.util.Timer;
        import java.util.TimerTask;
        public class Reminder{
                Timer timer;
                //构造子，在seconds秒后执行一个任务
                public Reminder(int seconds){
                        timer = new Timer();
                        timer.schedule(new RemindTask(),seconds*1000);  //schedule(清单
                }

                //内部成员类，描述将要执行的任务
```

```
        class RemindTask extends TimerTask{
                public void run(){
                        System.out.println("Time's up!");
                        timer.cancle();
                }
        }

        public static void main(String[] args){
                System.out.println("About to schedule task.");
                new Reminder(5);
                System.out.println("Task scheduled.");
        }

}
//Timer类是线程安全的，换言之，可以有多个线程共享同一个Timer对象，而不需要同步化。这是

//击鼓者（相当于客户端）
import java.lang.Thread;
import java.util.Timer;
import java.util.TimerTask;

public class DrumBeater{
        private static Player firstPlayer;
        public static boolean stopped = false;
        Timer timer;

        public static void main(String[] args){

                DrumBeater drumBeater = new DrumBeater();
                JiaMu jiaMu = new JiaMu(null);
                JiaMu.setSuccessor(new JiaShe(new JiaZheng(new JiaBaoYu(new Jia

                //开始击鼓过程
                drumBeater.startBeating(1);
                //由贾母开始传花
                firstPlayer = jiaMu;
                firstPlayer.handle();
```

```
                }

                //调用下面方法，开始击鼓过程
                public void startBeating(int stopInSeconds){
                        System.out.println("Drum beating started...");
                        timer = new Timer();
                        timer.schedule(new StopBeatingReminder(),stopInSeconds*1000);
                }

                //内部成员类，描述停止击鼓的任务
                class StopBeatingReminder extends TimerTask{
                        public void run(){
                                System.out.println("Drum beating stopped!");
                                stopped = true;
                                timer.cancel();
                        }
                }
        }

        //抽象传花者类
        public abstract class Player{
                public abstract void handle();

                private Player successor;

                public Player(){
                        successor = null;
                }

                protected void setSuccessor(Player aSuccessor){
                        successor = aSuccessor;
                }

                public void next(){
                        //判断下家对象是否有效
                        if(successor != null){
                                //将请求传给下家
```

```
                        successor.handle();
                }else{
                        //系统停止运行
                        System.out.println("Program is terminating.");
                        System.exit(0);
                }
        }
}

//贾母
public class JiaMu extends Player{
        public JiaMu(Player aSuccessor){
                this.setSuccessor(aSuccessor);
        }

        public void handle(){
                //检查是否击鼓已经停止 ]
                if(DrumBeater.stopped){
                        System.out.pritntln("Jia Mu gotta drink!");
                }else{
                        System.out.println("Jia Mu passed!");
                        next();
                }
        }
}
.....................
public class JiaHuan extends Player{
        public JiaHuan(Player aSuccessor){
                this.setSuccessor(aSuccessor);
        }

        public void handle(){
                //检查是否击鼓已经停止 ]
                if(DrumBeater.stopped){
                        System.out.println("Jia Huan gotta drink!");
                }else{
                        System.out.println("Jia Huan passed!");
```

```
                                next();
                            }
                        }
                    }
```

1.28 java设计模式笔记【行为模式第十二篇】

发表时间: 2009-10-23

状态（State Pattern）模式：又称为状态对象模式，状态模式是对象的行为模式
状态模式允许一个对象在其内部状态改变的时候改变其行为，这个对象看上去就像是改变了它的类一样。

模式所涉及的角色：
1、抽象状态角色：定义一个接口，用以封装环境对象的一个特定的状态所对应的行为。
2、具体状态角色：每一个具体状态类都实现了环境的一个状态所对应的行为。
3、环境角色：定义客户端所感兴趣的接口，并且保留一个具体状态类的实例。这个具体状态类的实例给出此环境的现有状态。

```
//环境角色
                public class Context{
                        private State state;

                        public void sampleOperation(){
                                state.sampleOperation();
                        }

                        public void setState(State state){
                                this.state = state;
                        }
                }

                //抽象状态角色
                public interface State{
                        void sampleOperation();
                }

                //具体状态类
                public class ConcreteState implements State{
                        public void sampleOperation(){}
                }
```

一、状态模式的效果：

1、状态模式需要对每一个系统可能取得的状态创立一个状态类的子类。当系统类的状态变化时，系统便改变所选的子类。

所有与一个特定的状态有关的行为都被包装到一个特定的对象里面，使得行为的定义局域化，因为同样的原因，如果有

新的状态以及对应的行为需要定义时，可以很方便地通过设立新的子类的方式加到系统里，不需要改动其他的类

2、由于每一个状态都被包装到子类里面，就可以不必采用过程性的处理方式，使用长篇累读的条件转移语句。

3、使用状态模式使 系统状态的变化变得很明显。由于不用一些属性（内部变量）来指明系统所处的状态，因此就不用担心

修改这些属性不当而造成的错误。

4、可以在系统的不同部分使用相同的一些状态类的对象。这种共享对象的办法是与亨元模式相符合的。事实上，此时这些状态

对象基本上是只有行为而没有内部状态的亨元模式。

5、状态模式的缺点是会造成大量的小的状态类：优点是使程序免于大量的条件转移语句，使程序实际上更易于维护。

6、系统所选的状态子类均是从一个抽象状态类或接口继承而来，java语言的特性，使得在java语言中使用状态模式较为安全。

多态性原则是状态模式的核心。

二、在什么情况下使用状态模式

1、一个对象的行为依赖于它所处的状态，对象的行为必须随着其状态的改变而改变。

2、对象在某个方法里依赖于一重或多重的条件转移语句，其中有大量的代码。状态模式把条件转移语句的每一个分支都包装到一个

单独的类里。这使得这些条件转移分支能够以类的方式独立存在和演化，维护这些独立的类也就不再影响到系统的其他部分。

三、状态模式实现时应该注意的地方

（1）谁来定义状态的变化。状态模式并没有规定哪一个角色决定状态发生转换的条件。如果转换条件是固定的，那么就可以把曲子存储

在编钟里，由编钟自己演奏，决定所发的音的转换。

然而，如果让State子类自行决定一它的下一个继任者是谁，以及在什么时候进行转换，就更灵活了。仍然以编钟为例，可以把

程序存储在每一个钟里，由每一个钟自行决定下面该发生的钟是哪一个，以及该在什么时候发声，这样就更有灵活性。

或者，由外界事件来决定状态的转换。也就是说，把编钟的演奏交给系统外部的演奏师。

（2）状态对象的创立和湮灭。其中值得权衡的做法有两种：

1、动态地创立所需要的状态对象，不要创立不需要的状态对象。当不再需要某一状态对象时，便马上把此状态对象的

实例湮灭掉。这相当于在需要某一只钟时，才把这只钟挂在编钟上；当这只钟使用完后，就立即把它卸下来；当再

次使用时，再把它挂回去，这对编钟来说不是一个好主意。

2、事先创立所有的对象，然后不再湮灭它们。这相当于把事先说的钟都挂在编钟上，然后在演奏过程中并不把它们卸

下来。这显然适用于状态变化比较快和濒繁、加载这些状态对象的成本较高的情况，比如编钟系统。

对于java系统，湮灭一个对象不是由程序完全控制的。当一个对象不再被引用时，java语言的垃圾处理器会自动把

它湮灭掉，但是具体在什么时候湮灭，则不是可以控制的，甚至不是可以预料的。

3、环境类把它自己作为参量传给状态对象。这样。一旦需要，状态对象就可以调用环境对象。

四、分派：

1、静态分派：发生在编译时期，分派根据静态类型信息发生，静态分派对于读者来说应当并不陌生，方法重载就是静态分派。

2、动态分派：发生在运行时期，动态分派动态地置换掉某个方法。面向对象的语言利用动态分派来实现方法置换产生的多态性。

1.29 java设计模式笔记【行为模式第十三篇】

发表时间: 2009-10-23

MVC模式：就是模型 - 视图 - 控制器模式

一、架构模式

1、模型端

在MVC模型里，模型便是执行某些任务的代码，而这部分代码并没有任何逻辑决定它对用户端的表示方法。

模型端只有纯粹的功能性接口，也就是一系列的公开方法。通过这些公开方法，便可以取得模型端的所有功能。

在这些公开方法中，有些是取值方法，让系统其他部分可以得到模型端的内部状态参数，其他的改值方法则允许

外部修改模型端的内部状态。

但是一般来说，模型端必须有方法登记视图，以便在模型端的内部状态发生变化时，可以通知视图端。

在java语言里，一个模型端可以继承java.util.Observerable类。此父类可以提供登记和通知视图所需的接口。

2、多个视图端

在MVC模式里面，一个模型端可以有几个视图端，而实际上复数的视图端是使用MVC的原始动机。

使用MVC模式可以允许多余一个的视图端存在，并且可以在需要的时候动态地登记上所需的视图。

在Excel表格中，一个饼图、一个棒图和一个表格均是同组数据的不同视图端，当用户通过任何一个视图修改数据时，

所有的视图都会按照新数据更新自己。

在java语言的java.awt库和javax.swing库里，所有的视窗构件均可以用来建造视图端。但是一个视图如果能够自动

得到更新，便需要实现java.util.Observer接口，这样便使得MVC模式符合观察者模式的定义。

在视图端里，视图可以嵌套，这意味着在视图端里均会有合成模式。

3、多个控制器端

MVC模式的视图端是与MVCF模式的控制器结合使用的。当用户端与相应的视图发生交互时，用户可以通过视窗更新模型的状态，

而这种更新是通过控制器端进行的。控制器端通过调用模型端的改值方法更改其状态值。与此同时，控制器端会通知所有的登记

了的视图刷新显示给用户。这意味着在视图端对象和控制器端对象之间会有观察者模式的应用。

一个控制器端对象在回应视图端请求时，会采用策略模式的方式决定如何回应。

如果想深入了解java模式，必须详细研读以上文章。必有所得啊。

虽然以上文章是我读书笔记，但是也是我的心得，希望大家尊重劳动成果，谢谢。

java苹果+番茄的博客文章

作者: java苹果+番茄
http://liulve-rover-163-com.javaeye.com