

adfgghj

博客园

首页

新随笔

联系

订阅

管理

随笔 - 5 文章 - 0 评论 - 1

昵称：ma_lihe
园龄：11个月
粉丝：2
关注：0
+加关注

< 2018年5月 >						
日	一	二	三	四	五	六
29	30	1	2	3	4	5
6	7	8	9	10	11	12
13	14	15	16	17	18	19
20	21	22	23	24	25	26
27	28	29	30	31	1	2
3	4	5	6	7	8	9

搜索

找找看

谷歌搜索

常用链接

我的随笔

我的评论

我的参与

最新评论

我的标签

我的标签

java(3)

内存(2)

虚拟机(2)

随笔分类

web(1)

多线程

设计模式(1)

算法(1)

虚拟机(2)

随笔档案

Java中23种设计模式--超快速入门及举例代码

在网上看了一些设计模式的文章后，感觉还是印象不太深刻，决定好好记录记录。
原文地址：<http://blog.csdn.net/doyumm2008/article/details/13288067>

注：本文代码基本都有很多没有初始化等等问题，主要是为了减少代码量，达到一眼就能了解大概情况的目的。

java的设计模式大体上分为三大类：


- 创建型模式（5种）：工厂方法模式，抽象工厂模式，单例模式，建造者模式，原型模式。
- 结构型模式（7种）：适配器模式，装饰器模式，代理模式，外观模式，桥接模式，组合模式，享元模式。
- 行为型模式（11种）：策略模式、模板方法模式、观察者模式、迭代子模式、责任链模式、命令模式、备忘录模式、状态模式、访问者模式、中介者模式、解释器模式。

设计模式遵循的原则有6个：

- 1、开闭原则（Open Close Principle）
对扩展开放，对修改关闭。
- 2、里氏代换原则（Liskov Substitution Principle）
只有当衍生类可以替换掉基类，软件单位的功能不受到影响时，基类才能真正被复用，而衍生类也能够在基类的基础上增加新的行为。
- 3、依赖倒转原则（Dependence Inversion Principle）
这个是开闭原则的基础，对接口编程，依赖于抽象而不依赖于具体。
- 4、接口隔离原则（Interface Segregation Principle）
使用多个隔离的借口来降低耦合度。
- 5、迪米特法则（最少知道原则）（Demeter Principle）
一个实体应当尽量少的与其他实体之间发生相互作用，使得系统功能模块相对独立。
- 6、合成复用原则（Composite Reuse Principle）
原则是尽量使用合成/聚合的方式，而不是使用继承。继承实际上破坏了类的封装性，超类的方法可能会被子类修改。

1. 工厂模式（Factory Method）

常用的工厂模式是静态工厂，利用static方法，作为一种类似于常见的工具类Utils等辅助效果，一般情况下工厂类不需要实例化。



```
interface food{}

class A implements food{}
class B implements food{}
class C implements food{}

public class StaticFactory {

    private StaticFactory(){}

    public static food getA(){ return new A(); }
```

2017年10月 (1)
2017年9月 (1)
2017年8月 (2)
2017年5月 (1)

最新评论

1. Re:Java中23种设计模式--超快速入门及举例代码

设计上的事就是这样，想到了，就能比较优雅的解决问题，想不到的话，就只能使用到处修改代码的方法比较笨拙的应对问题，还容易将项目弄的混乱。现在我比较庆幸当初学习了设计模式，而没有听其他人的“建议”，

--小猪-

阅读排行榜

1. Java中23种设计模式--超快速入门及举例代码(14266)

2. java内存模型详解(268)

3. 一篇文章彻底了解Java垃圾收集 (GC) 机制(101)

4. 最长回文子串--轻松理解Manacher算法(37)

5. WAMP3.1.10/Apache 设置站点根目录(14)

评论排行榜

1. Java中23种设计模式--超快速入门及举例代码(1)

```
public static food getB(){ return new B(); }
public static food getC(){ return new C(); }
}

class Client{
    //客户端代码只需要将相应的参数传入即可得到对象
    //用户不需要了解工厂类内部的逻辑。
    public void get(String name){
        food x = null ;
        if ( name.equals("A")) {
            x = StaticFactory.getA();
        }else if ( name.equals("B")){
            x = StaticFactory.getB();
        }else {
            x = StaticFactory.getC();
        }
    }
}
```

2. 抽象工厂模式 (Abstract Factory)

一个基础接口定义了功能，每个实现接口的子类就是产品，然后定义一个工厂接口，实现了工厂接口的就是工厂，这时候，接口编程的优点就出现了，我们可以新增产品类（只需要实现产品接口），只需要同时新增一个工厂类，客户端就可以轻松调用新产品的代码。

抽象工厂的灵活性就体现在这里，无需改动原有的代码，毕竟对于客户端来说，静态工厂模式在不改动StaticFactory类的代码时无法新增产品，如果采用了抽象工厂模式，就可以轻松的新增拓展类。

实例代码：

```
interface food{}

class A implements food{}
class B implements food{}

interface produce{ food get();}

class FactoryForA implements produce{
    @Override
    public food get() {
        return new A();
    }
}

class FactoryForB implements produce{
    @Override
    public food get() {
        return new B();
    }
}

public class AbstractFactory {
    public void ClientCode(String name){
        food x= new FactoryForA().get();
        x = new FactoryForB().get();
    }
}
```

3. 单例模式 (Singleton)

在内部创建一个实例，构造器全部设置为private，所有方法均在该实例上改动，在创建上要注意类的实例化只能执行一次，可以采用许多种方法来实现，如Synchronized关键字，或者利用内部类等机制来实现。

```
public class Singleton {
    private Singleton(){}

    private static class SingletonBuild{
        private static Singleton value = new Singleton();
    }
}
```

```
public Singleton getInstance(){ return SingletonBuild.value ;}

}
```

4.建造者模式 (Builder)

在了解之前，先假设有一个问题，我们需要创建一个学生对象，属性有name,number,class,sex,age,school等属性，如果每一个属性都可以为空，也就是说我们可以只用一个name,也可以用一个school,name,或者一个class,number，或者其他任意的赋值来创建一个学生对象，这时该怎么构造？

难道我们写6个1个输入的构造函数，15个2个输入的构造函数.....吗？这个时候就需要用到Builder模式了。给个例子，大家肯定一看就懂：

```
public class Builder {

    static class Student{
        String name = null ;
        int number = -1 ;
        String sex = null ;
        int age = -1 ;
        String school = null ;

        //构建器，利用构建器作为参数来构建Student对象
        static class StudentBuilder{
            String name = null ;
            int number = -1 ;
            String sex = null ;
            int age = -1 ;
            String school = null ;
            public StudentBuilder setName(String name) {
                this.name = name;
                return this ;
            }

            public StudentBuilder setNumber(int number) {
                this.number = number;
                return this ;
            }

            public StudentBuilder setSex(String sex) {
                this.sex = sex;
                return this ;
            }

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

            public StudentBuilder setSchool(String school) {
                this.school = school;
                return this ;
            }

            public Student build() {
                return new Student(this);
            }
        }

        public Student(StudentBuilder builder){
            this.age = builder.age;
            this.name = builder.name;
            this.number = builder.number;
            this.school = builder.school ;
            this.sex = builder.sex ;
        }

        public static void main( String[] args ){
            Student a = new Student.StudentBuilder().setAge(13).setName("LiHua").build();
            Student b = new Student.StudentBuilder().setSchool("sc").setSex("Male").setName("ZhangSan").build();
        }
    }
}
```



5. 原型模式 (Prototype)

原型模式就是讲一个对象作为原型，使用clone()方法来创建新的实例。



```
public class Prototype implements Cloneable{

    private String name;

    public String getName() {
        return name;
    }

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

    @Override
    protected Object clone() {
        try {
            return super.clone();
        } catch (CloneNotSupportedException e) {
            e.printStackTrace();
        } finally {
            return null;
        }
    }

    public static void main ( String[] args){
        Prototype pro = new Prototype();
        Prototype pro1 = (Prototype)pro.clone();
    }
}
```



此处使用的是浅拷贝，关于深浅拷贝，大家可以另行查找相关资料。

6. 适配器模式 (Adapter)

适配器模式的作用就是在原来的类上提供新功能。主要可分为3种：

- 类适配：创建新类，继承源类，并实现新接口，例如

```
class adapter extends oldClass implements newFunc{}
```

- 对象适配：创建新类持源类的实例，并实现新接口，例如

```
class adapter implements newFunc { private oldClass oldInstance ;}
```

- 接口适配：创建新的抽象类实现旧接口方法。例如

```
abstract class adapter implements oldClassFunc { void newFunc();}
```

7. 装饰模式 (Decorator)

给一类对象增加新的功能，装饰方法与具体的内部逻辑无关。例如：



```
interface Source{ void method();}
public class Decorator implements Source{

    private Source source ;
    public void decotatel(){
        System.out.println("decorate");
    }
    @Override
    public void method() {
        decotatel();
        source.method();
    }
}
```



8.代理模式 (Proxy)

客户端通过代理类访问，代理类实现具体的实现细节，客户只需要使用代理类即可实现操作。

这种模式可以对旧功能进行代理，用一个代理类调用原有的方法，且对产生的结果进行控制。



```
interface Source{ void method();}

class OldClass implements Source{
    @Override
    public void method() {
    }
}

class Proxy implements Source{
    private Source source = new OldClass();

    void doSomething(){
    @Override
    public void method() {
        new Class1().Func1();
        source.method();
        new Class2().Func2();
        doSomething();
    }
}
```



9.外观模式 (Facade)

为子系统的一组接口提供一个一致的界面，定义一个高层接口，这个接口使得这一子系统更加容易使用。这句话是百度百科的解释，有点难懂，但是没事，看下面的例子，我们在启动停止所有子系统的时候，为它们设计一个外观类，这样就可以实现统一的接口，这样即使有新增的子系统subSystem4,也可以在不修改客户端代码的情况下轻松完成。



```
public class Facade {
    private subSystem1 subSystem1 = new subSystem1();
    private subSystem2 subSystem2 = new subSystem2();
    private subSystem3 subSystem3 = new subSystem3();

    public void startSystem(){
        subSystem1.start();
        subSystem2.start();
        subSystem3.start();
    }

    public void stopSystem(){
        subSystem1.stop();
        subSystem2.stop();
        subSystem3.stop();
    }
}
```



10.桥接模式 (Bridge)

这里引用下<http://www.runoob.com/design-pattern/bridge-pattern.html>的例子。Circle类将DrwaApi与Shape类进行了桥接，代码：



```
interface DrawAPI {
    public void drawCircle(int radius, int x, int y);
}

class RedCircle implements DrawAPI {
    @Override
    public void drawCircle(int radius, int x, int y) {
        System.out.println("Drawing Circle[ color: red, radius: "
            + radius + ", x: " + x + ", y: " + y + "]");
    }
}
```

```

}

class GreenCircle implements DrawAPI {
    @Override
    public void drawCircle(int radius, int x, int y) {
        System.out.println("Drawing Circle[ color: green, radius: "
            + radius + ", x: " + x + ", " + y + "]");
    }
}

abstract class Shape {
    protected DrawAPI drawAPI;
    protected Shape(DrawAPI drawAPI){
        this.drawAPI = drawAPI;
    }
    public abstract void draw();
}

class Circle extends Shape {
    private int x, y, radius;

    public Circle(int x, int y, int radius, DrawAPI drawAPI) {
        super(drawAPI);
        this.x = x;
        this.y = y;
        this.radius = radius;
    }

    public void draw() {
        drawAPI.drawCircle(radius,x,y);
    }
}

//客户端使用代码
Shape redCircle = new Circle(100,100, 10, new RedCircle());
Shape greenCircle = new Circle(100,100, 10, new GreenCircle());
redCircle.draw();
greenCircle.draw();

```

11.组合模式 (Composite)

组合模式是为了表示那些层次结构，同时部分和整体也可能是一样的结构，常见的如文件夹或者树。
举例：

```

abstract class component{}

class File extends component{ String filename;}

class Folder extends component{
    component[] files ; //既可以放文件File类，也可以放文件夹Folder类。Folder类下又有子文件或子文件夹。
    String foldername ;
    public Folder(component[] source){ files = source ;}

    public void scan(){
        for ( component f:files){
            if ( f instanceof File){
                System.out.println("File "+((File) f).filename);
            }else if(f instanceof Folder){
                Folder e = (Folder)f ;
                System.out.println("Folder "+e.foldername);
                e.scan();
            }
        }
    }
}

```

12.享元模式 (Flyweight)

使用共享对象的方法，用来尽可能减少内存使用量以及分享资讯。通常使用工厂类辅助，例子中使用一个HashMap类进行辅助判断，数据池中是否已经有了目标实例，如果有，则直接返回，不需要多次创建重复实例。

```

abstract class flywei{ }

public class Flyweight extends flywei{
    Object obj ;
    public Flyweight(Object obj){
        this.obj = obj;
    }
}

class FlyweightFactory{
    private HashMap<Object,Flyweight> data;

    public FlyweightFactory(){ data = new HashMap<>();}

    public Flyweight getFlyweight(Object object){
        if ( data.containsKey(object)){
            return data.get(object);
        }else {
            Flyweight flyweight = new Flyweight(object);
            data.put(object,flyweight);
            return flyweight;
        }
    }
}

```

分类: 设计模式

标签: [java](#)

好文要顶 关注我 收藏该文  

 [ma_lihe](#)
关注 - 0 0
粉丝 - 2 0
[+加关注](#)

» 下一篇: [java内存模型详解](#)

posted @ 2017-05-30 18:42 ma_lihe 阅读(14267) 评论(1) 编辑 收藏

评论列表

#1楼 2018-04-14 17:08 小猪- 回复 引用

设计上的事就是这样，想到了， 就能比较优雅的解决问题，想不到的话， 就只能使用到处修改代码的方法比较笨拙的应对问题，还容易将项目弄的混乱。现在我比较庆幸当初学习了设计模式，而没有听其他人的“建议”， 很多人都说“我们做的项目中用不到设计模式，学这个没用”。关于学习这个问题在我的另一篇博客 我为什么要学习Linux?中提到过。设计模式是个好东西，以后我肯定还会进一步的学习，并且在项目中多实践，提升自己的设计能力。当然也可以建议建议你们看看这套设计模式视频 <https://pan.baidu.com/s/1dTCdog> 密码：29oc 或许会给你们一些启发
其实设计模式并不难，难的是真正领悟他的精妙，并且能灵活的运用于日常项目的开发。
支持(0) 反对(0)

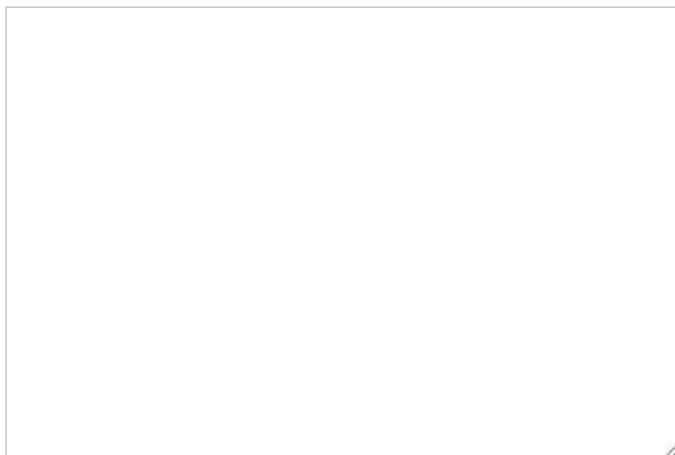
发表评论

刷新评论 刷新页面 返回顶部

昵称：

评论内容：





[提交评论](#) [退出](#) [订阅评论](#)

[Ctrl+Enter快捷键提交]

【推荐】超50万VC++源码：大型组态工控、电力仿真CAD与GIS源码库！

【活动】2050 大会 - 博客园程序员团聚（5.25 杭州·云栖小镇）

【推荐】0元免费体验华为云服务

【活动】腾讯云招募自媒体，共享百万资源包



最新IT新闻：

- Nature机器学习子刊封闭式访问遭Jeff Dean等大牛联合抵制
 - 苹果股票回购规模增千亿美元 季度红利上调16%至每股73美分
 - FB开发者大会：自黑的小扎、不买账的开发者，和消失的黑科技
 - 上海地铁正式引入语音购票 未来还能刷脸进站
 - 小米联合Oculus发布VR一体机Oculus Go：中国版夏季开售
- » 更多新闻...



最新知识库文章：

- 如何识别别人的技术能力和水平？
 - 写给自学者的入门指南
 - 和程序员谈恋爱
 - 学会学习
 - 优秀技术人的管理陷阱
- » 更多知识库文章...