单列模与工厂模式

什么是设计模式
被反复使用，
最常用的一种设计模式

设计模式是优秀的使用案例，使用设计模式可提高代码的重用性，让代码更容易理解
保证代码可靠

总体来说设计模式分为三大类：

创建型模式，共五种：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，共七种：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，共十一种：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。

其实还有两类：并发型模式和线程池模式。用一个图片来整体描述一下：

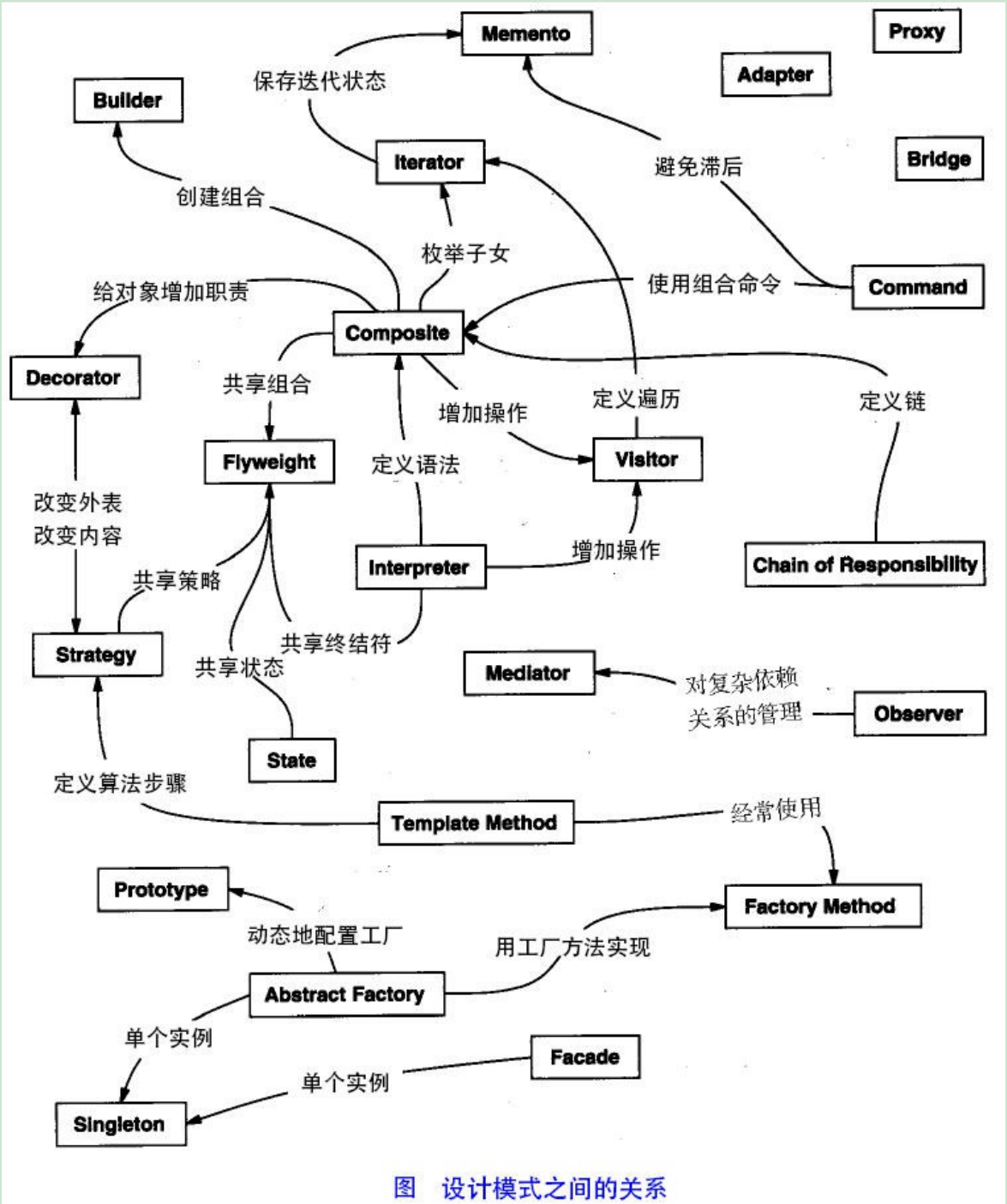


图　设计模式之间的关系

很多时候我们只需要一个，实例，如果创造多个实例，就会导致问题，比如占用资源过多，不一致的结果

单例模式的作用，就是保证整个应用程序中某一个实例，有且只有一个，
类型：饿汉模式，懒汉模式
饿汉模式： 当Singleton这个类被加载的时候，它就会去创建一个类的唯一实例，不管我们是否需要这个实例，他都已经加载了，只有类加载了，它就加载了

代码实现方式

主要还是通过构造方法私有化

```
public class Singleton {
//1 将构造方法私有化，允许外部直接创建对象
private  Singleton(){

   }
//2 创建类的实例，唯一实例
private  static Singleton intance=new Singleton();
      //3 提供一个用于获取实例的方法，使用public static修饰
    public static Singleton getIntance() {
return intance;
}
```

饿汉模式

```
public class singleton2 {

private  singleton2(){

    }

public static singleton2 getInstance() {
return instance;
    }

private   static singleton2 instance = new singleton2();
}
```

懒汉模式

```
/*懒汉模式*/
public class Singleton2 {
//1  将构造方法私有化，不允许外边直接创建对象
private  Singleton2(){

    }

//2 创建类的唯一实例  private  static 修饰
private static  Singleton2 instance;

//3 提供一个用于获取实例的方法，使用public static 修饰
public static Singleton2 getinstance(){
if(instance==null){
instance =new Singleton2();
}
return instance;
}
}
```

```
public class Test {

public static void main(String[] args) {
//饿汉模式
Singleton s1= Singleton.getIntance();
Singleton s2= Singleton.getIntance();
System.out.println(s1==s2);//true
     //懒汉模式
Singleton2 s3=Singleton2.getinstance();
Singleton2 s4=Singleton2.getinstance();
```

```
System.out.println(s3==s4);
}
}
```

区别：饿汉模式的特点加载类时比较慢，但运行时获取对象速度比较快，线程安全
懒汉模式相比 特点加载类比较快，但是在运行时获取对象的速度比较慢，线程不安全
线程不安全是因为，没有在懒汉模式那里加一个同步块，而且没有加同步块和再加一个判断的话，这里的懒汉模式并不是真正意义上的单例模式

工厂模式

实例化对象：用**工厂方法代替new**操作。

工厂模式包括工厂方法模式和抽象工厂模式

抽象工厂模式是工厂方法模式的**扩展**

工厂模式的意图

定义一个接口来创建对象，但是让子类来决定哪些类**需要被实例化**

工厂方法**把实例化的工作推迟到子类去实现**

什么情况下时候工厂模式

有一组**类似的对象**需要创建

在编码时**不能预见需要**创建哪种类的实例

系统需要考虑**扩展性**，**不应依赖**于产品实例如何被创建。组合和表达的细节

项目中的现状：(低耦合)

在软件系统中经常面临着“对象”的创建工作，由于需求的变化，这个对象随之也会发生变化，但它拥有比较稳定的接口

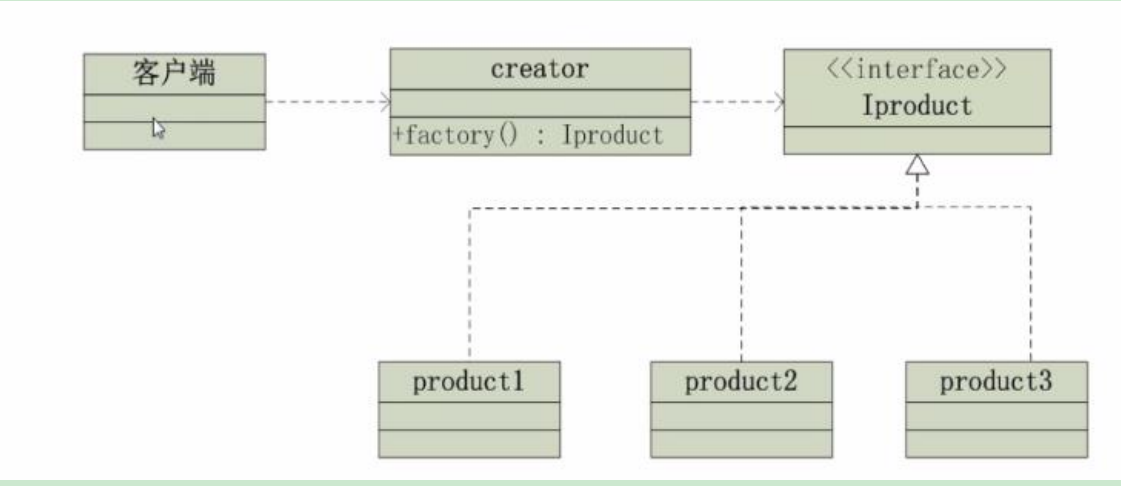
为此：我们需要提供一种封装机制来隔离出这个易变对象的变化，从而保持系统中其他依赖该对象不随着需求变化而变化

基于项目现状将代码进行如下设计：

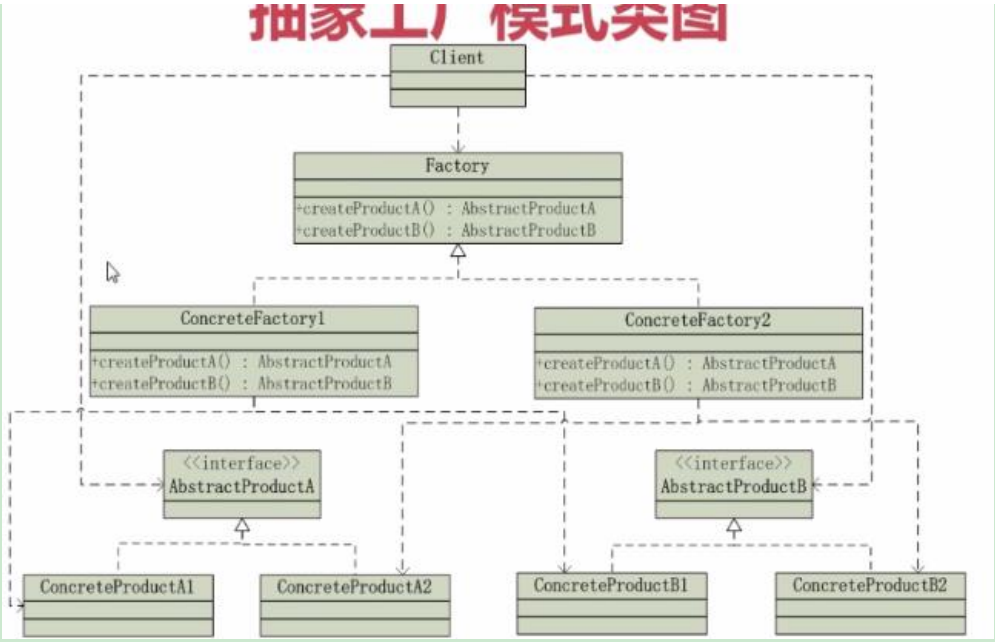
1 尽量低耦合，一个对象的依赖关系的变化与本身无关

2具体产品与客户端分离，责任分割

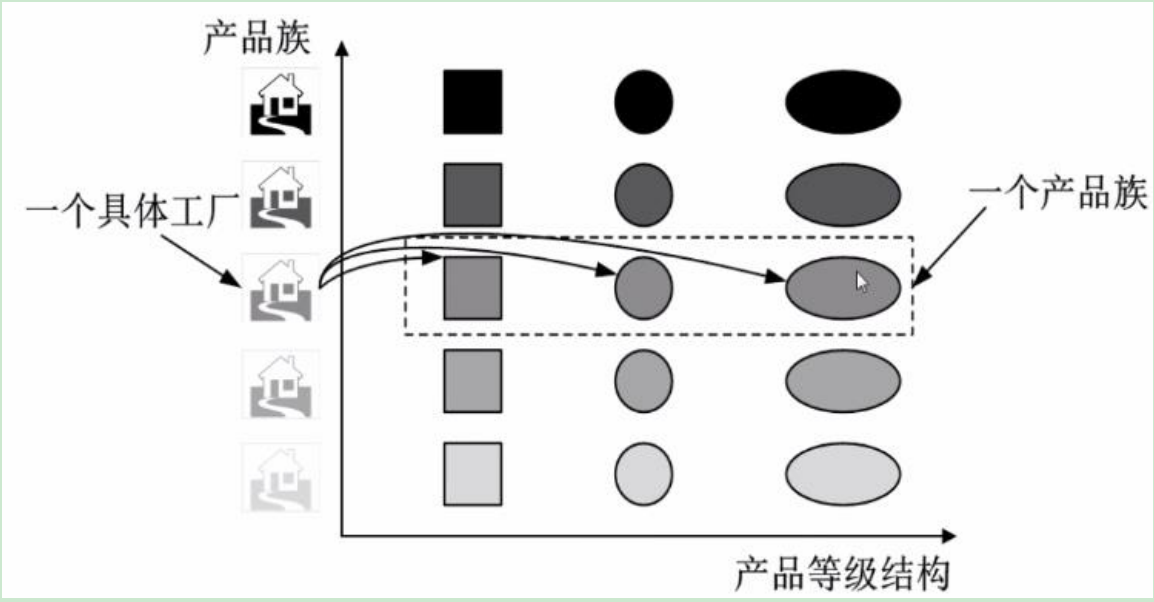
工厂模式：



抽象工厂模式：



是系列与家族一种关系



通过反射模式可以创建对象，还有通过类型工厂也可以创建对象

下面的代码是简单工厂模式

http://blog.csdn.net/jason0539/article/details/23020989这里有简单工厂与工厂模式的介绍与具体结构

```
package FactoryModel;

/**
 * Created by Administrator on 2016/4/20.
 *//*模拟客户端的实现*/
public class SunyTest {
public static void main(String[] args) {
//          HairInterface left= new leftHair();
//          left.draw();

HariFactory factory=new HariFactory();
//         HairInterface left=factory.getHair("left");
//         left.draw();
//
//         HairInterface right=factory.getHair("right");
//         right.draw();
//          HairInterface left=factory.getHairByClass("FactoryModel.leftHair");
//          left.draw();

HairInterface left=factory.getHairByClass("left");
left.draw();
HairInterface in=factory.getHairByClass("in");
```

```
in.draw();
}

}
```

```
package FactoryModel;

import java.util.Map;

/**
 * Created by Administrator on 2016/4/20.
 *//*发型工厂*/
public class HariFactory {
/*根据类型工厂来创建对象*/
public  HairInterface getHair(String key){
if("left".equals(key)){
return  new leftHair();
}else if("right".equals(key)){
return  new RightHair();
}
return  null;
}
/*
*  根据类的名称来生产对象
*  这是通过反射*/
public  HairInterface getHairByClass(String className)   {
try {
          Map<String,String>map=new  PropertiesReader().getProperties();

HairInterface hair= (HairInterface) Class.forName(map.get(className)).newInstance();
          return hair;
} catch (Exception e) {
          e.printStackTrace();
}
return  null;
}

}
```

```
package FactoryModel;

/**
 * Created by Administrator on 2016/4/20.
 */
public class leftHair implements  HairInterface {
/*画了一个左偏风发型*/
@Override
public void draw() {
        System.out.println("----------左偏风发型-------------");
}
}
```

```
package FactoryModel;

/**
 * Created by Administrator on 2016/4/20.
 */
/*右偏风发型*/
public class RightHair implements  HairInterface {


@Override
public void draw() {
        System.out.println("-------右偏风发型---------");

}
}
```

```
package FactoryModel;

/**
 * Created by Administrator on 2016/4/20.
 */
/**
 * 中分发型
```

```
 * @author Administrator
 *
 */
public class InHair implements HairInterface {

@Override
public void draw() {
// TODO Auto-generated method stub
System.out.println("------------------中分发型--------------------");

}

}
```

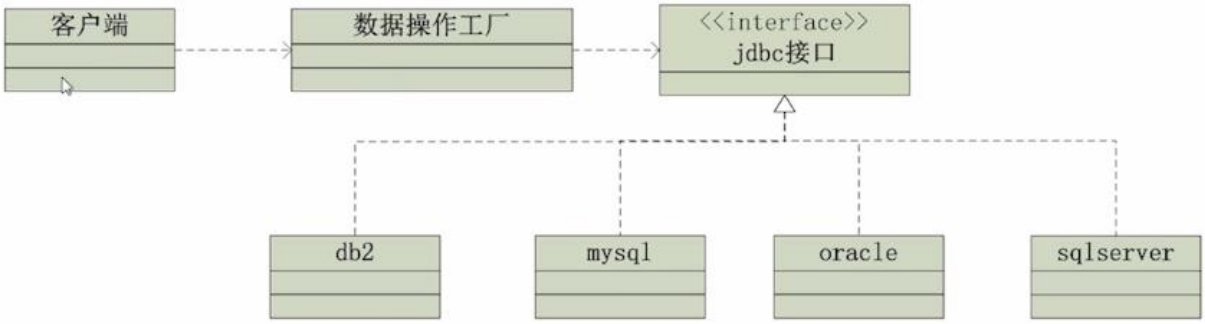
```
package FactoryModel;

/**
 * Created by Administrator on 2016/4/20.
 */
public interface HairInterface {

public  void draw();
}
```

● JDBC

是一种用于执行SQL语句的Java API，可以为多种关系数据库提供统一访问，它由一组用Java语言编写的类和接口组成。



• spring beanfactory

BeanFactory ，作为 Spring 基础的 IoC 容器，是 Spring 的一个 Bean 工厂。

如果单从工厂模式的角度思考，它就是用来“生产 Bean ”，然后提供给客户端。

Bean的实例化过程如下：

- 调用Bean的默认构造方法，或指定的构造方法，生成bean实例（暂称为instance1）
- 如果Bean的配置文件中注入了Bean属性值，则在instance1基础上进行属性注入形成instance2，这种注入是覆盖性的
- 如果Bean实现了InitializingBean接口，则调用afterPropertiesSet()方法，来改变或操作instance2，得到instance3

- 如果Bean的配置文件中指定了init-method="init"属性，则会调用指定的初始化方法，则在instance3的基础上调用初始化方法init()，将对象最终初始化为instance4；当然，这个初始化的名字是任意的

工厂方法模式和抽象工厂模式对比

- 工厂模式是一种极端情况的抽象工厂模式，而抽象工厂模式可以看成是工厂模式的推广
- 工厂模式用来创建一个产品的等级结构，而抽象工厂模式是用来创建多个产品的等级结构
- 工厂模式只有一个抽象产品类，而抽象工厂模式有多个抽象产品类

工厂模式的实现帮助我们

- 系统可以在不修改具体工厂角色的情况下引进新的产品
- 客户端不必关心对象如果创建，明确了职责
- 更好的理解面向对象的原则 面向接口编程，而不要面向实现编程

工厂模式适用与哪些场景

- 一个系统应当不依赖于产品类实例被创立，组成，和表示的细节。这对于所有形态的工厂模式都是重要的
- 这个系统的产品有至少一个的产品族
- 同属于同一个产品族的产品是设计成在一起使用的。这一约束必须得在系统的设计中体现出来
- 不同的产品以一系列的接口的面貌出现，从而使系统不依赖于接口实现的细节

好吧，我发现抽象工厂模式代码太多，就不粘贴了
http://www.cnblogs.com/java-my-life/archive/2012/03/28/2418836.html 网站在这
http://design-patterns.readthedocs.io/zh_CN/latest/creational_patterns/abstract_factory.html#abstract-factory