设计模式

本文是对大话设计模式一书的笔记（持续更新，直到读完）

1. 简单工厂

简单工厂不属于23种设计模式，但是工厂模式和抽像工厂模式都是由此演化而来。**简单工厂就是由一个工厂类决定创建出哪一种具体实例**，而不像用户暴露具体细节。

```
class ProductFactory{
    public static Product CreateProduct(String p){
        Product product;
        switch(p){
            case "A":
                product = new ProductA;
                break;
            case "B":
                product = new ProductB;
                break;
            case "C":
                product = new ProductC;
                break;
        }
        return product;
    }
}
```

2.工厂模式

核心思想：在执行过程中的类需要可替换。多态中利用接口可以实现类传入类的可替换。比如我们有一个接口，有一个函数接收这个接口类型的参数。这样我们写这个函数的程序，并不需要已经写好具体实现接口的类。只在调用的时候传入合适的类即可。
工厂模式则是这种思路的延续，只不过传入参数不是产品类，而是工厂类，到里面运行create函数拿到具体类。这样做效果和原来一样。不过工厂不一定只生产一种产品，也就是说可能有createProduct1,2,3,4等等方法。
实例：spring的FactoryBean

```java
//抽象产品
public interface Product {
    String productName();
}

//具体产品1
public class ConcreteProduct1 implements Product {

    @Override
    public String productName() {
        return "This is ConcreteProduct1";
    }
}

//具体产品2
public class ConcreteProduct2 implements Product {

    @Override
    public String productName() {
        return "This is ConcreteProduct2";
    }
}

//抽象工厂
public interface Factory {
    Product CreateProduct(String name);
}

//具体工厂
public class ConcreteFactory1 implements Factory{

    @Override
    public Product CreateProduct(String name) {
        if("ConcreteProduct1".equals(name)){
            return new ConcreteProduct1();
        }else if("ConcreteProduct2".equals(name)){
            return new ConcreteProduct2();
        }else {
            return null;
        }
    }
}

//for test
public class Main {
    public static void main(String[] args) {
        Factory factory = new ConcreteFactory1();
        Product concreteProduct1 = factory.CreateProduct("ConcreteProduct1");
        Product concreteProduct2 = factory.CreateProduct("ConcreteProduct2");
        System.out.println(concreteProduct1.productName());
        System.out.println(concreteProduct2.productName());
```

```
    }
}
```

3.抽象工厂

进一步抽象，就是再建工厂的问题

4.单例模式

1)饿汉式：不管你需不需要，new了再说
缺点：浪费空间

```
public class Singleton {
    private static Singleton instance = new Singleton(); //直接new
    private Singleton (){}
    public static Singleton getInstance() {
        return instance;
    }
}
```

2)懒汉式：我比较懒，你需要我再new
缺点：每次调用 getInstance() 都需要加锁，影响性能

```
//线程安全版本,加锁
public class Singleton {
    private static Singleton instance;//只声明，不new
    private Singleton (){}
    public static synchronized Singleton getInstance() {
        if (instance == null) {
            instance = new Singleton();
        }
        return instance;
    }
}
```

3)静态内部类：静态内部类在JVM中是唯一的，可以保证单例对象的唯一性

```
public class Singleton{
    private static class SingletonHolder{
        private static final Singleton INSTANCE = new Singleton;
    }
    private Singleton(){}
    public static final Singleton getInstance(){
        return SingletonHolder.Instance;
    }
}
```

4)双重校验锁: 双锁模式在懒汉式的基础上做了优化，给静态对象的定义加上volatile来保证初始化对象的唯一性，在获取对象时通过 syschronized(Singleton.class) 来给单例类加锁，保证操作的唯一性

```
public class Singleton {
    //保证可见性和指令重排
    private volatile static Singleton instance; //1.对象锁
    //私有构造函数
    private Singleton(){}
    public static Singleton getInstance(){
        //第一重检查，可以避免每次调用都需要加锁
        if(instance == null){
            //同步锁定代码块
            synchronized (Singleton.class){ //2.synchronized方法锁
                //第二重检查
                if(instance == null){
                    //注意：非原子操作
                    instance = new Singleton();
                }
            }
        }
        return instance;
    }
}
```

5. 策略模式（Strategy）

策略一般指的是算法部分，将策略封装成一个成员变量放入执行类中，在不同的情况下注入不同的策略
核心思想：将策略（算法）抽象成一个接口，实现这个策略接口生成各种具体策略，从而注入不同的实现类。
应用场景：ribbon的负载均衡策略的替换（如随机，均匀）；JPA中@Id的生成策略（如递增）

```
//策略Strategy API
public interface Stratery{
    public abstract void Algorithm();
}
```

```
//具体策略 ConcreteStrategy
class ConcreteStrategyA implements Stratery{
    @override
    public void Algorithm(){
        System.out.println("algorithm A implements");
    }
}

class ConcreteStrategyB implements Stratery{
    @Override
    public void Algorithm(){
        System.out.println("algorithm B implements");
    }
}


//上下文Context
class Context{
    Strategy strategy;

    public Context(Strategy strategy){
        this.strategy = strategy;
    }

    public void ContextAlgorithm(){
        strategy.Algorithm();
    }

}


//客户端
Class client{
    public static void main(String[] args){
        Context context;

        context = new Context(Strategy StrategyA);
        context.ContextAlgorithm();

        context = new Context(Strategy StrategyB);
        context.ContextAlgorithm();
    }
}
```

6.代理模式