在这部分主要阅读了github上，王洋先生写的各种设计模式的源代码。

针对每一种设计模式都给出了相应的代码设计方法，讲解，以及真正的代码展示。

源代码链接为：<https://github.com/wangyang135/DesignPatterns>

同步博客为：<http://blog.csdn.net/wangyang1354/>

首先参考王洋本人写的博客将集中设计模式的几个原则摘抄如下：

1. 单一功能原则（Single responsibility principle）规定每个类都应该有一个单一的功能，并且该功能应该由这个类完全封装起来。所有它的（这个类的）服务都应该严密的和该功能平行（功能平行，意味着没有依赖）。
2. 里氏替换原则，“派生类（子类）对象能够替换其基类（超类）对象被使用。” 以上内容并非利斯科夫的原文，而是译自罗伯特·马丁（Robert Martin）对原文的解读。
3. 对隔离原则的理解是: 设计接口的时候，尽量保证实现接口的那些类尽可能一致的包含着接口中的方法，避免过多的设计了接口中的方法，导致其实现类中需要实现多个完全没有用处的方法（会造成代码的冗余和混乱）。
4. 最少知道法则：面向对象的程序设计中，对象与对象之间尽量相互独立，具体对象的行为由具体的对象去完成，而不是由某个对象去指定另一个对象去实施行为而且是具体的行为。迪米特法则，核心的思想就是，要求我们在设计的时候，尽量避免类与类之间的耦合
5. 开闭原则：开闭原则相当于一个纲领性质的原则，提倡类等应该在设计完成后通过扩展的方式适应新的业务需求，而不是通过修改的方式去适应新的需求，这样的设计更加灵活、稳定。之前的五大原则是开闭原则思想的具体实现的情况。

**其中主要了解的设计模式是，工厂模式：**

# 简单工厂模式

简单工厂模式其实并不算是一种设计模式，更多的时候是一种编程习惯。

**定义：**定义**一个**工厂类，根据传入的参数不同返回不同的实例，被创建的实例具有共同的父类或接口。

**使用场景：**其实由定义也大概能推测出其使用场景，首先由于只有一个工厂类，所以工厂类中创建的对象不能太多，否则工厂类的业务逻辑就太复杂了，其次由于工厂类封装了对象的创建过程，所以客户端应该不关心对象的创建。总结一下适用场景：  
　　（1）需要创建的对象较少。  
　　（2）客户端不关心对象的创建过程。  
 以上就是简单工厂模式简单工厂模式的适用场景，下面看一个具体的实例。

**实例：**

创建一个可以绘制不同形状的绘图工具，可以绘制圆形，正方形，三角形，每个图形都会有一个draw()方法用于绘图，不看代码先考虑一下如何通过该模式设计完成此功能。

由题可知圆形，正方形，三角形都属于一种图形，并且都具有draw方法，所以首先可以定义一个接口或者抽象类，作为这三个图像的公共父类，并在其中声明一个公共的draw方法。

public interface Shape {

void draw();

}

这里定义成抽象类也是可以的，只不过接口是更高一级的抽象，所以习惯定义成接口，而且接口支持多实现，方便以后扩展。

下面就是编写具体的图形，每种图形都实现Shape 接口

圆形

public class CircleShape implements Shape {

public CircleShape() {

System.out.println( "CircleShape: created");

}

public void draw() {

System.out.println( "draw: CircleShape");

}

}

正方形

public class RectShape implements Shape {

public RectShape() {

System.out.println( "RectShape: created");

}

@Override

public void draw() {

System.out.println( "draw: RectShape");

}

}三角形

public class TriangleShape implements Shape {

public TriangleShape() {

System.out.println( "TriangleShape: created");

}

@Override

public void draw() {

System.out.println( "draw: TriangleShape");

}

}

下面是工厂类的具体实现

public class ShapeFactory {

public static final String TAG = "ShapeFactory";

public static Shape getShape(String type) {

Shape shape = null;

if (type.equalsIgnoreCase("circle")) {

shape = new CircleShape();

} else if (type.equalsIgnoreCase("rect")) {

shape = new RectShape();

} else if (type.equalsIgnoreCase("triangle")) {

shape = new TriangleShape();

}

return shape;

}

}

　　在这个工厂类中通过传入不同的type可以new不同的形状，返回结果为Shape 类型，这个就是简单工厂核心的地方了。  
客户端使用

画圆形

Shape shape= ShapeFactory.getShape("circle");

shape.draw();

画正方形

Shape shape= ShapeFactory.getShape("rect");

shape.draw();

画三角形

Shape shape= ShapeFactory.getShape("triangle");

shape.draw();　　只通过给ShapeFactory传入不同的参数就实现了各种形状的绘制。

**工厂方法模式**

　　工厂方法模式是简单工厂的仅一步深化， 在工厂方法模式中，我们不再提供一个统一的工厂类来创建所有的对象，而是针对不同的对象提供不同的工厂。也就是说每个对象都有一个与之对应的工厂。

**定义：**

　　定义一个用于创建对象的接口，让子类决定将哪一个类实例化。工厂方法模式让一个类的实例化延迟到其子类。

　　这次我们先用实例详细解释一下这个定义，最后在总结它的使用场景。

**实例：**

　　现在需要设计一个这样的图片加载类，它具有多个图片加载器，用来加载jpg，png，gif格式的图片，每个加载器都有一个read（）方法，用于读取图片。下面我们完成这个图片加载类。

首先完成图片加载器的设计，编写一个加载器的公共接口。

public interface Reader {

void read();

}

Reader 里面只有一个read（）方法，然后完成各个图片加载器的代码。

**Jpg图片加载器**

public class JpgReader implements Reader {

@Override

public void read() {

System.out.print("read jpg");

}

}

**Png图片加载器**

public class PngReader implements Reader {

@Override

public void read() {

System.out.print("read png");

}

}

**Gif图片加载器**

public class GifReader implements Reader {

@Override

public void read() {

System.out.print("read gif");

}

}

现在我们按照定义所说定义一个抽象的工厂接口ReaderFactory

public interface ReaderFactory {

Reader getReader();

}

里面有一个getReader（）方法返回我们的Reader 类，接下来我们把上面定义好的每个图片加载器都提供一个工厂类，这些工厂类实现了ReaderFactory 。

**Jpg加载器工厂**

public class JpgReaderFactory implements ReaderFactory {

@Override

public Reader getReader() {

return new JpgReader();

}

}

**Png加载器工厂**

public class PngReaderFactory implements ReaderFactory {

@Override

public Reader getReader() {

return new PngReader();

}

}

**Gif加载器工厂**

public class GifReaderFactory implements ReaderFactory {

@Override

public Reader getReader() {

return new GifReader();

}

}

在每个工厂类中我们都通过复写的getReader（）方法返回各自的图片加载器对象。

**客户端使用**

读取Jpg

ReaderFactory factory=new JpgReaderFactory();

Reader reader=factory.getReader();

reader.read();

读取Png

ReaderFactory factory=new PngReaderFactory();

Reader reader=factory.getReader();

reader.read();

读取Gif

ReaderFactory factory=new GifReaderFactory();

Reader reader=factory.getReader();

reader.read();

　　可以看到上面三段代码，分别读取了不同格式的图片，不同之处在于针对不同的图片格式声明了不同的工厂，进而创建了相应的图片加载器。

　　通过这个实例各位小伙伴是不是对工厂模式有了进一步的理解呢，和简单工厂对比一下，最根本的区别在于，简单工厂只有一个统一的工厂类，而工厂方法是针对每个要创建的对象都会提供一个工厂类，这些工厂类都实现了一个工厂基类（本例中的ReaderFactory ）。下面总结一下工厂方法的适用场景。

**适用场景：**

　　（1）客户端不需要知道它所创建的对象的类。例子中我们不知道每个图片加载器具体叫什么名，只知道创建它的工厂名就完成了床架过程。

　　（2）客户端可以通过子类来指定创建对应的对象。

以上场景使用于采用工厂方法模式。

**抽象工厂模式**

　　这个模式最不好理解，而且在实际应用中局限性也蛮大的，因为这个模式并不符合开闭原则。实际开发还需要做好权衡。

　　抽象工厂模式是工厂方法的仅一步深化，在这个模式中的工厂类不单单可以创建一个对象，而是可以创建一组对象。这是和工厂方法最大的不同点。

**定义：**

　　提供一个创建一系列相关或相互依赖对象的接口，而无须指定它们具体的类。（ 在抽象工厂模式中，每一个具体工厂都提供了多个工厂方法用于产生多种不同类型的对象）

抽象工厂和工厂方法一样可以划分为4大部分：

AbstractFactory（抽象工厂）声明了一组用于创建对象的方法，注意是一组。

ConcreteFactory（具体工厂）：它实现了在抽象工厂中声明的创建对象的方法，生成一组具体对象。

AbstractProduct（抽象产品）：它为每种对象声明接口，在其中声明了对象所具有的业务方法。

ConcreteProduct（具体产品）：它定义具体工厂生产的具体对象。

下面还是先看一个具体实例。

**实例：**

　　现在需要做一款跨平台的游戏，需要兼容Android，Ios，Wp三个移动操作系统，该游戏针对每个系统都设计了一套操作控制器（OperationController）和界面控制器（UIController），下面通过抽闲工厂方式完成这款游戏的架构设计。

　　由题可知，游戏里边的各个平台的UIController和OperationController应该是我们最终生产的具体产品。所以新建两个抽象产品接口。

抽象操作控制器

public interface OperationController {

void control();

}

抽象界面控制器

public interface UIController {

void display();

}

然后完成各个系统平台的具体操作控制器和界面控制器

Android

public class AndroidOperationController implements OperationController {

@Override

public void control() {

System.out.println("AndroidOperationController");

}

}

public class AndroidUIController implements UIController {

@Override

public void display() {

System.out.println("AndroidInterfaceController");

}

}

Ios

public class IosOperationController implements OperationController {

@Override

public void control() {

System.out.println("IosOperationController");

}

}

public class IosUIController implements UIController {

@Override

public void display() {

System.out.println("IosInterfaceController");

}

}

Wp

public class WpOperationController implements OperationController {

@Override

public void control() {

System.out.println("WpOperationController");

}

}

public class WpUIController implements UIController {

@Override

public void display() {

System.out.println("WpInterfaceController");

}

}

下面定义一个抽闲工厂，该工厂需要可以创建OperationController和UIController

public interface SystemFactory {

public OperationController createOperationController();

public UIController createInterfaceController();

}

在各平台具体的工厂类中完成操作控制器和界面控制器的创建过程

Android

public class AndroidFactory implements SystemFactory {

@Override

public OperationController createOperationController() {

return new AndroidOperationController();

}

@Override

public UIController createInterfaceController() {

return new AndroidUIController();

}

}

Ios

public class IosFactory implements SystemFactory {

@Override

public OperationController createOperationController() {

return new IosOperationController();

}

@Override

public UIController createInterfaceController() {

return new IosUIController();

}

}

Wp

public class WpFactory implements SystemFactory {

@Override

public OperationController createOperationController() {

return new WpOperationController();

}

@Override

public UIController createInterfaceController() {

return new WpUIController();

}

}

客户端调用：

SystemFactory mFactory;

UIController interfaceController;

OperationController operationController;

//Android

mFactory=new AndroidFactory();

//Ios

mFactory=new IosFactory();

//Wp

mFactory=new WpFactory();

interfaceController=mFactory.createInterfaceController();

operationController=mFactory.createOperationController();

interfaceController.display();

operationController.control();

针对不同平台只通过创建不同的工厂对象就完成了操作和UI控制器的创建。

适用场景：

（1）和工厂方法一样客户端不需要知道它所创建的对象的类。

（2）需要一组对象共同完成某种功能时。并且可能存在多组对象完成不同功能的情况。

（3）系统结构稳定，不会频繁的增加对象。（因为一旦增加就需要修改原有代码，不符合开闭原则）