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。

**1、开闭原则（Open Close Principle）**

开闭原则就是说**对扩展开放，对修改关闭**。在程序需要进行拓展的时候，不能去修改原有的代码（工厂方法与简单工厂），实现一个热插拔的效果。所以一句话概括就是：为了使程序的扩展性好，易于维护和升级。想要达到这样的效果，我们需要使用接口和抽象类，后面的具体设计中我们会提到这点。

**2、里氏代换原则（Liskov Substitution Principle）**

里氏代换原则(Liskov Substitution Principle LSP)面向对象设计的基本原则之一。 里氏代换原则中说，任何基类可以出现的地方，子类一定可以出现。 LSP是继承复用的基石，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。里氏代换原则是对“开-闭”原则的补充。实现“开-闭”原则的关键步骤就是抽象化。而基类与子类的继承关系就是抽象化的具体实现，所以里氏代换原则是对实现抽象化的具体步骤的规范。—— From Baidu 百科

**3、依赖倒转原则（Dependence Inversion Principle）**

这个是开闭原则的基础，具体内容：真对接口编程，依赖于抽象而不依赖于具体。

**4、接口隔离原则（Interface Segregation Principle）**

这个原则的意思是：使用多个隔离的接口，比使用单个接口要好。还是一个降低类之间的耦合度的意思，从这儿我们看出，其实设计模式就是一个软件的设计思想，从大型软件架构出发，为了升级和维护方便。所以上文中多次出现：降低依赖，降低耦合。

**5、迪米特法则（最少知道原则）（Demeter Principle）**

为什么叫最少知道原则，就是说：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。

**其他如：Ioc**

**单例模式**

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实例。  
3、单例类必须给所有其他对象提供这一实例。

1. **public** **static** Singleton getInstance() {
2. **if** (singleton == **null**) {
3. **synchronized** (Singleton.**class**) {
4. **if** (singleton == **null**) {
5. singleton = **new** Singleton();
6. }
7. }
8. }
9. **return** singleton;
10. }
11. **public** **class** Singleton {
12. **private** **static** **class** LazyHolder {
13. **private** **static** **final** Singleton INSTANCE = **new** Singleton();
14. }
15. **private** Singleton (){}
16. **public** **static** **final** Singleton getInstance() {
17. **return** LazyHolder.INSTANCE;
18. }
19. }

//线程安全，不需同步

工厂模式：

作用：

目的是为了解耦，还可以隐藏类名，也不需要准备构造方法的参数。

简单new一个对象，如果许多地方都有要修改时则都需要修改并编译。

几种实现方式

参见17.2

代理

Class simpleProxy{

Private Interface proxied;

Public SimpleProxy(Interface proxied){

This.proxied = proxied;

}

Public void doSomethid(){

Proxied.doSomethind();

}

}

动态代理

Interface **proxy** = (Interface)Proxy.**newProxyInstance**(

Interface.class.getClassLoader();//**一个类加载器**

New Class[]{Interface.class}//**代理实现的列表接口 ，这里只希望代理Interface**

New DynamicProxyHandler(real)) //InvocationHandler的一个实现);

Proxy.doSomething()

**InvocationHandler**主要实现 **invoke**(Object proxy,Method method,Object [] args)和它的构造器

//实现

构造方法(要代理的对象){

Proxied = 要代理的对象;

}

invoke(Object proxy,Method method,Object [] args){

method.invoke(proxied,args);

}

MVC

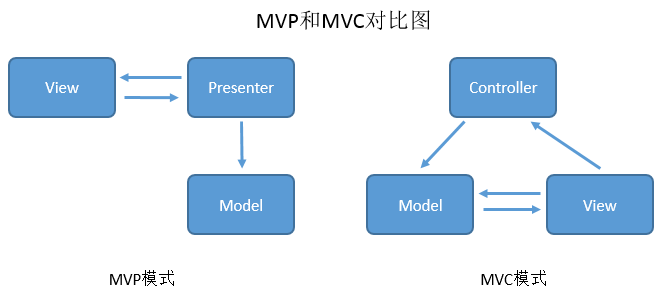
模型(model)－视图(view)－控制器(controller)的缩写

Model（模型）是应用程序中用于处理应用程序数据逻辑的部分。 通常模型对象负责在数据库中存取数据。

View（视图）是应用程序中处理数据显示的部分。

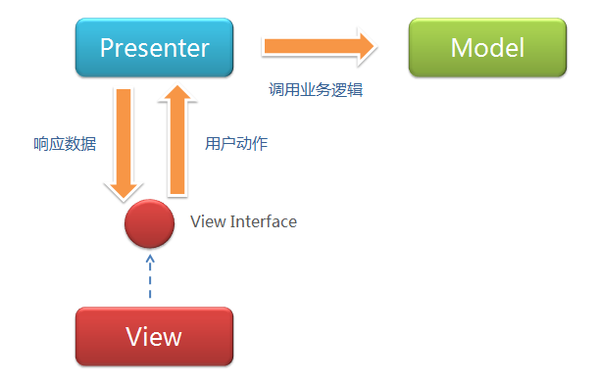
Controller（控制器）是应用程序中处理用户交互的部分。 通常控制器负责从视图读取数据，控制用户输入，并向模型发送数据。

MVC的一般流程是这样的：View（界面）触发事件--》Controller（业务）处理了业务，然后触发了数据更新--》不知道谁更新了Model的数据--》Model（带着数据）回到了View--》View更新数据（非实际应用中的mvc）



MVP:（实际开发中 mvc和mvp一致）

MVP是把MVC中的Controller换成了Presenter（呈现），目的就是为了完全切断View跟Model之间的联系，由Presenter充当桥梁，做到View-Model之间通信的完全隔离。



Models分为ps、ds和dao

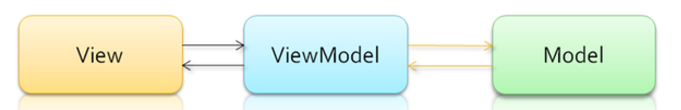
Controller层:Controller层负责具体的业务模块流程的控制

Dao层持久化

Service分为ds和ps，ps针对单一接口来调用ds层的具体业务逻辑实现方法。Ds层可由不同接口复用。

MVVM

Model-View-ViewModel



ViewModel大致上就是MVP的Presenter和MVC的Controller了，而View和ViewModel间没有了MVP的界面接口，而是直接交互，用数据“绑定”的形式让数据更新的事件不需要开发人员手动去编写特殊用例，而是**自动**地双向同步。

在MVVM中，数据和业务逻辑处于一个独立的View Model中，ViewModel只要关注数据和业务逻辑，不需要和UI或者控件打交道。由数据自动去驱动UI去自动更新UI，UI的改变又同时自动反馈到数据，数据成为主导因素，这样使得在业务逻辑处理只要关心数据，方便而且简单很多。