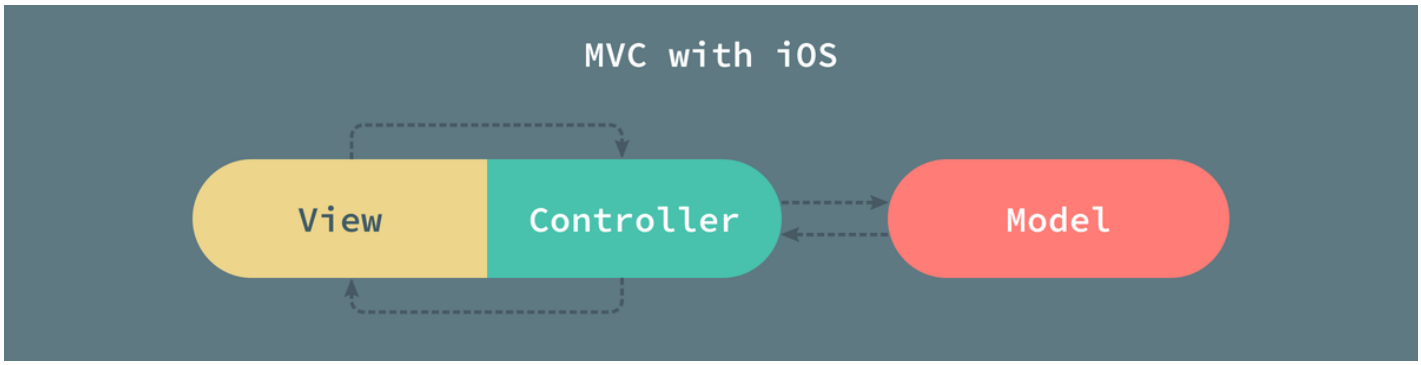
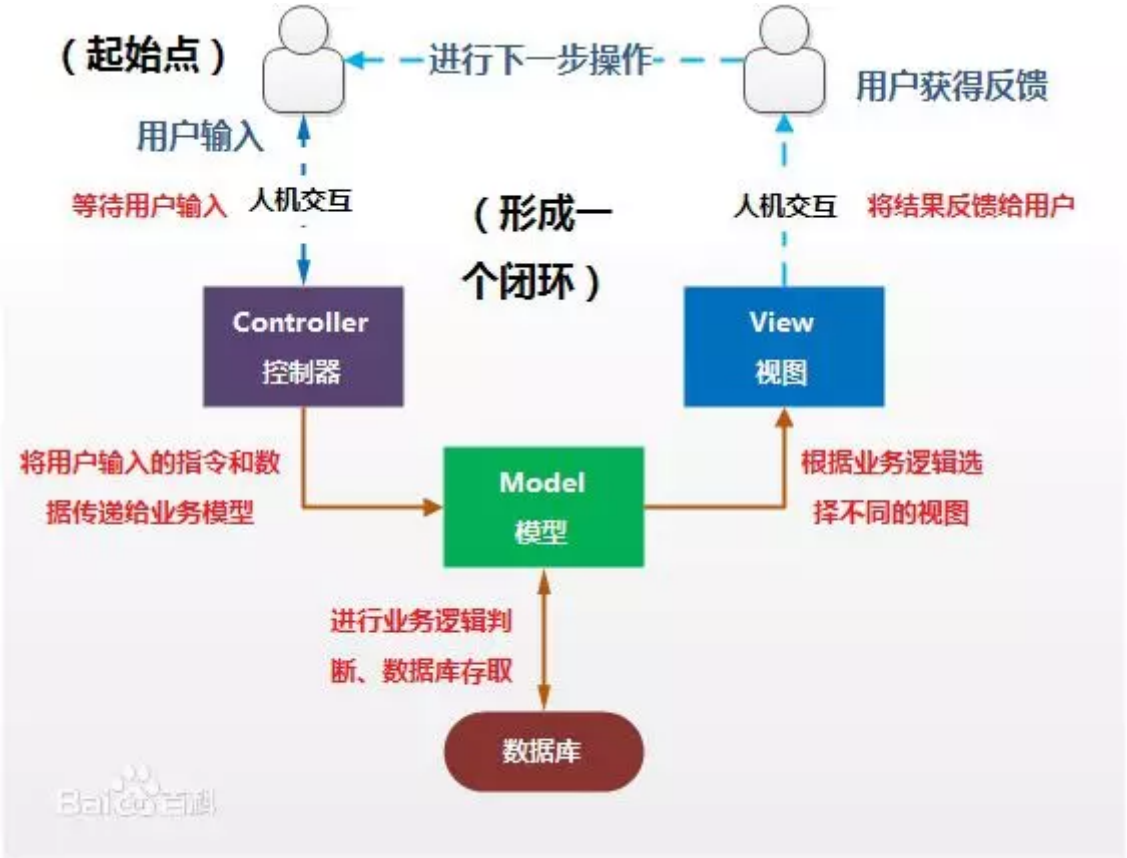
# 设计模式

## MVC框架理解




模型(model)－视图(view)－控制器(controller)的缩写
一种软件设计典范，用一种业务逻辑、数据、界面显示分离的方法组织代码，将业务逻辑聚集到一个部件里面，在改进和个性化定制界面及用户交互的同时，不需要重新编写业务逻辑。MVC被独特的发展起来用于映射传统的输入、处理和输出功能在一个逻辑的图形化用户界面的结构中。

- model通常是读写数据的部分（容器，数据结构）；
- view是应用程序展示数据的部分；
- controller则处理用户交互，收发数据等。



## 单一职责原则

## 里氏替换原则

只要有父类出现的地方，子类就能替代出现，使用者根本不必知道是父类还是子类，不会引起错误和异常。（但是反过来就未必了。）
比如cs里枪支引进玩具枪，玩具枪具备枪支的形状、声音特性，但是不具备杀敌的特性，如果不能完全实现父类的业务，最好把玩具枪从枪支里抽离出来。
采用里氏替换原则的目的就是增强程序健壮性，即使增加子类，原有的子类还可以继续运行。

## 依赖倒置原则

高层模块不应该依赖低层模块，两者都应该依赖其抽象，抽象不应该依赖细节，细节应该依赖抽象。
面向接口编程。
“采用依赖倒置原则可以减少类间的耦合性，提高系统的稳定性，降低并行开发引起的风险，提高代码的可读性和可维护性。”
对象依赖有三种传递方式：

- 构造函数传递依赖对象
- setter方法传递依赖对象
- 接口声明依赖对象

依赖倒置原则的本质就是通过抽象使得各个类之间实现彼此独立、互不影响，实现模块之间的松耦合。
如何在项目中实践这个规则？

- 每个类尽量都有接口或抽象类
- 变量的表面尽量是接口或者抽象类
- 任何类都不应该从具体类派生
- 尽量不要重写基类的方法
- 结合里氏替换原则

## 接口隔离原则

接口尽量细化，同时接口中的方法尽可能少，而不是建立起庞大臃肿的接口，容纳所有客户端的访问。
比如选美女，可以把脸蛋、形态和气质分离成两个独立的接口，这样可以预防未来变更的扩散，提高系统的灵活性和可维护性。

- 接口要尽量小
- 接口要高内聚，提高处理能力，减少对外交互
- 对不同人群能够定制服务（比如图书查询权限）
- 接口设计是有限度的，接口颗粒度越小越灵活，但是这也会给工程带来结构的复杂化，可维护性降低

## 迪米特法则

LoD，也称最小知识原则(LKP)，一个对象应该对其他对象有最少的了解。一个类应该对自己需要耦合或调用的类知道的最少，你内部如何复杂跟我没有关系，那是你的事情，我就知道你提供的这么多的public方法，其余我一概不关心。
一个类公开的public属性或方法越多，修改时涉及的面就越大，变更引起的风险扩散也就越大，因此在设计的阶段需要反复权衡是否还可以减少public属性和方法。

## 开闭原则

一个软件实体如类、模块和函数应该对扩展开放，对修改关闭。
比如书店卖书，出现变化：对书本按价格分类打折处理。

- 修改接口？这种做法破坏了接口的稳定性，接口不应该经常发生变化
- 修改实现类？会影响到想看原价的用户，导致信息不对称
- 通过扩展实现变化？增加一个新子类，重写getPrice的方法，修改少、风险也小

1.抽象约束

- 通过接口或抽象类约束扩展，对扩展进行边界设定，不允许出现在接口或抽象类中不存在的public方法
- 参数类型、引用对象尽量使用接口或者抽象类，而不是实现类
- 抽象层尽量保持稳定，一旦确定及不允许修改

2.元数据（metadata）控制模块行为
3.制定项目章程，制定所有成员都必须遵守的约定
4.封装变化

## 单例模式

确保一个类只有一个实例，而且实例自行实例化并向系统提供这个实例

```
1  public class Singleton{
2      private static final Singleton singleton = new Singleton();
3  // 限制产生多个对象
4      private Singleton(){
5      }
6  // 提供该方法获得实例对象
7      public static Singleton getSingleton(){
8          return singleton;
9      }
10 // 类中其他方法尽量用static
11     public static void doSomething{
```

```
    }
}
```

- 优点
  - 由于单例模式只存在一个实例，减少了内存开支，特别是一个对象需要频繁创建、销毁时
  - 减少系统的性能开销，当一个对象的产生需要比较多的资源时，通过在应用启动时产生一个单例对象，然后用永久驻留内存的方式来解决
  - 避免对资源的多重占用，避免对同一个资源的同时写操作
  - 可以在系统设置全局的访问点，例如可以涉及一个单例类，负责所有数据表的映射管理
- 缺点
  - 一般没有接口，扩展困难。因为接口对于单例模式是没有任何意义的，它被要求自行实例化，并提供单一实例
  - 对测试不利，如果单例模式没有完成，不能进行测试
  - 单例模式与单一职责冲突，一个类应该实现一个逻辑，而不关心它是否是单例的
- 使用环境
  - 要求生成唯一序列号的环境
  - 在整个项目中需要一个共享访问点或共享数据，如web计数器，使用单例模式保持计数器的值，并保证线程安全
  - 创建一个资源需要消耗的资源过多
  - 需要定义大量的静态常量和方法（当然，也可以直接声明为static）
- 注意事项
  - 高并发情况下的线程同步问题，比如以下代码

```
public class C{
    public static C c = null;
    private C(){}
    public static C getC(){
        if(c == null) c = new C();
        return c;
    }
}
```

  - 不要复制单例类
- 扩展：两三个对象（用个标记值计算数量，用容器存储）

## 代理模式（委托模式）

为其他对象提供一种代理以控制这个对象的访问

- 抽象主题角色：可以是抽象类也可以是接口，是一个普通业务类型定义
- 具体主题角色：被代理角色，是业务的具体执行者
- 代理主题角色：它负责对真实角色的应用，把所有抽象主题都委托给真是主题角色实现

```
public interface Subject{
    public void request();
}
public static RealSubject implements Subject{
    public void request(){
    }
}
public class Proxy implements Subject{
    private Subject subject = null;
    public Proxy(){
        this.subject = new Proxy();
    }
    public Proxy(Object...objects){}
    public void request(){
        this.before();
        this.subject().request();
        this.after();
    }
    private void before(){}
    private void after(){}
}
```

- 优点
  - 职责清晰，真实的角色就是实现实际的业务逻辑，不用关心其他非本职责的事务，通过后期的代理完成一件事物，使得编程简洁清晰
  - 高拓展性，具体角色随时会变化，只要它实现了接口，代理类就能完全不做任何修改的情况下使用
  - 智能化
- 使用场景
  - 比如打官司，你不想参与中间过程的是是非非，只要完成自己的答辩就成，其他的事情交给律师搞定
- 扩展

- 普通代理：客户端只能访问代理角色，不能访问真实角色，屏蔽了真实角色对高层影响，适合扩展性高的场合
- 强制代理：必须通过真实角色找到代理，否则不能访问，由真实角色指定代理角色
- 动态代理：实现阶段不用关心代理谁，而在运行阶段才指定代理哪一个对象（感觉现在用不上，先跳过了）

- 示例 婴儿吃饭睡觉

## 观察者模式

定义对象间一对多的依赖关系，使得每当一个对象改变状态，则所有依赖于它的对象都会收到通知。

- Subject被观察者：能够动态地增加、取消观察者。必须实现的指责：管理观察者并通知观察者。
- Observer观察者：观察者接到消息后，对信息进行处理。
- ConcreteSubject具体的被观察者：定义自己的业务逻辑，同时定义对哪些事件进行通知。
- ConcreteObserver具体的观察者：定义自己的处理逻辑。

- 优点
  - 观察者和被观察者之间是抽象耦合（更容易扩展）
  - 建立一套触发机制

    根据单一职责原则，实现的类是单一指责的，有的业务逻辑会把事件形成一个触发链。观察者模式可以完美地实现这里的链条形式。

- 缺点
  - 开发和运行过程中的效率问题

- 注意事项
  - 广播链：如果一个对象既是观察者，又是被观察者，逻辑就会比较复杂，可维护性非常差。
  - 异步处理：被观察者发生动作，观察者要作出回应，如果观察者数量庞大，处理时间较长，可以考虑异步处理。

## 工厂模式

关于这个设计模式，光看《设计模式之禅》简直把我看晕了。

浅显易懂的通过OC介绍工厂模式

相对绕了点，但总结的不错