**设计模式**

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。

设计模式的六大原则：

总原则－开闭原则

对扩展开放，对修改封闭。在程序需要进行拓展的时候，不能去修改原有的代码，而是要扩展原有代码，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等，后面的具体设计中我们会提到这点。

1、单一职责原则

不要存在多于一个导致类变更的原因，也就是说每个类应该实现单一的职责，否则就应该把类拆分。

2、里氏替换原则（Liskov Substitution Principle）

任何基类可以出现的地方，子类一定可以出现。里氏替换原则是继承复用的基石，只有当衍生类可以替换基类，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。里氏替换原则中，子类对父类的方法尽量不要重写和重载。因为父类代表了定义好的结构，通过这个规范的接口与外界交互，子类不应该随便破坏它。

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

面向接口编程，依赖于抽象而不依赖于具体。写代码时用到具体类时，不与具体类交互，而与具体类的上层接口交互。

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

每个接口中不存在子类用不到却必须实现的方法，如果不然，就要将接口拆分。使用多个隔离的接口，比使用单个接口（多个接口方法集合到一个的接口）要好。

5、迪米特法则（最少知道原则）（Demeter Principle）

一个类对自己依赖的类知道的越少越好。无论被依赖的类多么复杂，都应该将逻辑封装在方法的内部，通过public方法提供给外部。这样当被依赖的类变化时，才能最小的影响该类。

最少知道原则的另一个表达方式是：只与直接的朋友通信。类之间只要有耦合关系，就叫朋友关系。耦合分为依赖、关联、聚合、组合等。我们称出现为成员变量、方法参数、方法返回值中的类为直接朋友。局部变量、临时变量则不是直接的朋友。我们要求陌生的类不要作为局部变量出现在类中。

6、合成复用原则（Composite Reuse Principle）

尽量首先使用合成/聚合的方式，而不是使用继承。

**·装饰模式**：动态地给一个对象添加一些额外的职责。就增加功能来说，装饰模式相比生成子类更为灵活。

使用场景：

● 需要扩展一个类的功能，或给一个类增加附加功能。

● 需要动态地给一个对象增加功能，这些功能可以再动态地撤销。

● 需要为一批的兄弟类进行改装或加装功能，当然是首选装饰模式。

**·适配器模式（Adapter Pattern）**：将一个类的接口变换成客户端所期待的另一种接口，从而使原本因接口不匹配而无法在一起工作的两个类能够在一起工作。

使用场景：

你有动机修改一个已经投产中的接口时，适配器模式可能是最适合你的模式。比如系统扩展了，需要使用一个已有或新建立的类，但这个类又不符合系统的接口，怎么办？使用适配器模式，这也是我们例子中提到的。

注意事项：

详细设计阶段不要考虑使用适配器模式，使用主要场景为扩展应用中。

对象适配器：

 对象适配器和类适配器的区别：

类适配器是类间继承，对象适配器是对象的合成关系，也可以说是类的关联关系，这是两者的根本区别。（实际项目中对象适配器使用到的场景相对比较多）。

**·代理模式（Proxy Pattern）：**为其他对象提供一种代理以控制对这个对象的访问。

● Subject抽象主题角色

抽象主题类可以是抽象类也可以是接口，是一个最普通的业务类型定义，无特殊要求。

● RealSubject具体主题角色

也叫做被委托角色、被代理角色。它才是冤大头，是业务逻辑的具体执行者。

● Proxy代理主题角色

也叫做委托类、代理类。它负责对真实角色的应用，把所有抽象主题类定义的方法限制委托给真实主题角色实现，并且在真实主题角色处理完毕前后做预处理和善后处理工作。

普通代理和强制代理：

普通代理就是我们要知道代理的存在，也就是类似的GamePlayerProxy这个类的存在，然后才能访问；

强制代理则是调用者直接调用真实角色，而不用关心代理是否存在，其代理的产生是由真实角色决定的。

普通代理：

在该模式下，调用者只知代理而不用知道真实的角色是谁，屏蔽了真实角色的变更对高层模块的影响，真实的主题角色想怎么修改就怎么修改，对高层次的模块没有任何的影响，只要你实现了接口所对应的方法，该模式非常适合对扩展性要求较高的场合。

强制代理：

强制代理的概念就是要从真实角色查找到代理角色，不允许直接访问真实角色。高层模块只要调用getProxy就可以访问真实角色的所有方法，它根本就不需要产生一个代理出来，代理的管理已经由真实角色自己完成。

动态代理：

根据被代理的接口生成所有的方法，也就是说给定一个接口，动态代理会宣称“我已经实现该接口下的所有方法了”。

两条独立发展的线路。动态代理实现代理的职责，业务逻辑Subject实现相关的逻辑功能，两者之间没有必然的相互耦合的关系。通知Advice从另一个切面切入，最终在高层模块也就是Client进行耦合，完成逻辑的封装任务。

动态代理的意图：横切面编程，在不改变我们已有代码结构的情况下增强或控制对象的行为。

首要条件：被代理的类必须要实现一个接口。

**·单例模式（Singleton Pattern）**：确保某一个类只有一个实例，而且自行实例化并向整个系统提供这个实例。

使用场景：

● 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，例如一个Web页面上的计数器，可以不用把每次刷新都记录到数据库中，使用单例模式保持计数器的值，并确保是线程安全的；

● 创建一个对象需要消耗的资源过多，如要访问IO和数据库等资源；

● 需要定义大量的静态常量和静态方法（如工具类）的环境，可以采用单例模式（当然，也可以直接声明为static的方式）。

**·工厂模式：**定义一个用于创建对象的接口，让子类决定实例化哪一个类。工厂方法使一个类的实例化延迟到其子类。

Product为抽象产品类负责定义产品的共性，实现对事物最抽象的定义；

Creator为抽象创建类，也就是抽象工厂，具体如何创建产品类是由具体的实现工厂ConcreteCreator完成的。

简单工厂模式：

一个模块仅需要一个工厂类，没有必要把它产生出来，使用静态的方法

多个工厂类：

每个人种（具体的产品类）都对应了一个创建者，每个创建者独立负责创建对应的产品对象，非常符合单一职责原则

代替单例模式：

单例模式的核心要求就是在内存中只有一个对象，通过工厂方法模式也可以只在内存中生产一个对象

延迟初始化：

ProductFactory负责产品类对象的创建工作，并且通过prMap变量产生一个缓存，对需要再次被重用的对象保留

使用场景：jdbc连接数据库，硬件访问，降低对象的产生和销毁