学习内容：  
1. 常用的设计模式和使用场景

任务  
1. 给出单例模式、观察者模式、建造者模式、工厂模式实现demo

**设计模式**

**单例模式**

使用内部类的方式实现单例，既可以做到延迟加载，也不必使用同步关键字，是一种比较完善的实现。

序列化和反序列化可能会破坏单例。

常用场景：创建的一个对象需要消耗的资源过多，比如 I/O 与数据库的连接

案例地址：https://github.com/ben201708/learn\_java/tree/master/src/com/company/design\_mode/single

**代理模式**

代理模式：为其他对象提供一种代理以控制对这个对象的访问。

使用代理模式的意图有很多，比如因为安全原因，需要屏蔽客户端直接访问真实对象；或者在远程调用中，需要使用代理类处理远程方法调用的细节技术（如RMI）；也可能是为了提升系统性能，对真实对象进行封装，从而达到延迟加载的目的。

**代理模式角色**

|  |  |
| --- | --- |
| 角色 | 作用 |
| 主题接口 | 定义代理类和真实主题的公共对外方法，也是代理类代理真实主题的方法。 |
| 真实主题 | 真正实现业务逻辑的类 |
| 代理类 | 用来代理和封装真实主题 |
| Main | 客户端，使用代理类和主题接口完成工作 |

延迟加载的核心思想：如果当前并没有使用这个组件，则不需要真正的初始化它，使用一个代理对象替代它的原有的位置，只要在真正需要使用的时候，才会加载它。使用代理模式的延迟加载是非常有意义的，首先，它可以在时间轴上分散系统的压力，尤其在启动时，不必要完成所有初始化操作，从而加速启动时间；其次对很多真实主题，在软件启动到关闭可能根本不会被调用到，初始化这些数据无疑是一种资源浪费。

常用场景： 远程代理；虚拟代理；Cache代理

案例地址：https://github.com/ben201708/learn\_java/tree/master/src/com/company/design\_mode/proxy

**简单工厂模式&工厂方法模式**

简单工厂模式的最大优点在于工厂类中包含了必要的逻辑判断，根据客户端的选择条件动态的实例化相关的类，对于客户端来说，去除了与具体产品的依赖，但是却未被了开放-封闭原则。

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

常用场景：数据库访问，当用户不知道最后系统采用哪一类数据库，以及数据库可能有变化时；

案例地址：https://github.com/ben201708/learn\_java/tree/master/src/com/company/design\_mode/factory

**抽象工厂模式**

抽象工厂模式(Abstract Factory)，提供一个创建一系列相关或相互依赖对象的接口，而无需指定他们具体的类。

抽象工厂好处：①便于交换产品系列，由于具体工厂类，在一个应用中只需要在初始化的时候出现一次，这就使得改变一个应用的具体工厂变得非常容易，它值需要改变具体工厂即可使用不同的产品配置了。②让具体的创建实例过程与客户端分离，客户端是通过他们的抽象接口操纵实例，产品的具体类名也被具体工厂的实现分离，不会出现在客户代码中。

缺点：涉及到改动到底层类时，需要改动对应的继承工厂的所有子类，改动量比较大。

缺点优化解决方法：用反射（可以去除switch语句）+抽象工厂模式+（xml获取当前工厂类配置文件）。Assembly.Load(“程序集名称”.).CreateInstance(“命名空间.类名称”)

使用场景：生成不同操作系统的程序

**策略模式**

策略模式是一种定义一系列算法的方法，从概念上看，所有这些算法完成的都是相同的工作，只是实现不同，他可以以相同的方式调用所有算法，减少各种算法与使用算法类之间的耦合。

当不同的行为堆砌在一类中时，就很难避免使用条件语句来选择合适的行为。将这些行为封装在一个个独立的Strage类中，可以使用这些行为的勒种消除条件语句。

策略模式就是用来封装算法的，但是在实践中，我们发现可以用它来封装几乎任何类型的规则，只要在分析过程中听到需要在不同时间运用不同的业务规则，就可以考虑使用策略模式处理这种变化的可能性。

在基本的策略模式中，选择所用具体实现的职责由客户端对象承担，并传给策略模式的Context对象。这本身并没有接触客户端需要选择判断的压力，而策略模式与简单工厂模式结合，选择具体实现的职责也可以由Context来承担，这就最大化的减轻客户端的职责。

使用场景：商场不同时间点使用不同降价模型；

案例地址：https://github.com/ben201708/learn\_java/tree/master/src/com/company/design\_mode/strategy

**装饰模式**

动态的给一个对象添加一些额外的职责，就增加功能来说，装饰模式比生成子类更加灵活。

当系统需要新功能的时候，是向旧的类中添加新的代码，这些新加的代码通常装饰了原有类的核心职责或者主要行为。由于在主类中加入新字段、新逻辑，增主类的复杂度，而这些新加入的逻辑仅仅为了满足一些只在特定情况下才会执行的特殊行为的需要。装饰模式却提供了很好的解决方法，它把每一个要装饰的功能放在单独的类中，并让这个类包装它索要装饰的对象，因此，当需要执行特殊行为时，客户端就可以在运行是更加需要有选择地、按顺序地使用装饰功能包装对象。

装饰模式优点：把类中的装饰功能从类中搬移去除，这样可以简化原有的类，有效的把类的核心职责和装饰功能区分开，去除相关类的重复的装饰逻辑。

使用场景：动态扩展一个类的功能

案例地址：https://github.com/ben201708/learn\_java/tree/master/src/com/company/design\_mode/decorator

**组合模式**

**组合模式**（Composite），将对象组合成树形结构以表示“部分-整体”的层次结构。组合模式使得用户对单个对象和组合对象的使用具有一致性。

**透明方式**：在Component中声明所有用来管理子对象的方法，其中包括add，remove等。这样实现Component接口的所有子类都具备了add和remove，这样做的好处是叶节点和枝节点对于外界没有区别，它们具备完全一致的行为接口。但问题也很明显，因为Leaf类本身不具备add和remove方法的功能，所以实现它是没有意义的。

**安全方式**：在Component接口中不去声明add和remove方法，那么子类的Leaf就不需要去实现它，而是在Composite声明所有用来改哪里子类对象的方法，不过这样由于不够透明，所以树叶和树枝类将不具有相同的接口，客户端的调用需要做相应的判断，带来了不便。

**何时使用组合模式：**当需求中体现部分与整体层次的结构时，以及你希望用户可以忽略组合对象与单个对象的不同，统一地使用组合结构中的所有对象时，就应该考虑用组合模式。

使用场景：公司部门树形结构；树形菜单、文件或者文件夹等

组合模式案例：https://github.com/ben201708/learn\_java/tree/master/src/com/company/design\_mode/composite

**适配者模式**

适配器模式（Adapter），将一个类的接口转换成客户希望的另外一个接口。Adapter模式使得原本由于接口不兼容二不能一起工作的那些类可以一起工作。

系统的数据和行为都正确，但接口不符时，应该考虑使用适配器，目的是使控制范围之外的一个原有对象与某个接口匹配。适配器模式主要应用于希望复用一些现存的类，但是接口又与复用环境要求不一致的情况。

DataAdapter用作DataSet和数据源之间的适配器以便检索和保存数据。DataAdapter通过映射Fill(这更改了DataSet中的数据以便与数据源中的数据匹配)和Update来提供这一适配器。

使用场景：修改兼容系统新老接口

案例地址：https://github.com/ben201708/learn\_java/tree/master/src/com/company/design\_mode/adapter

**观察者模式**

观察者模定义了一种一对多的依赖关系，让多个观察者对象同时监听某一个主题对象。这个主题对象在状态发生变化时，会通知所有观察者对象，使他们能够自动更新自己。

将一个系统分割成一系列相互协作的类有一个很不好的副作用，那就是需要维护相关对象见的一致性。我们不希望为了维护一致性而使各类紧密耦合，这样会给维护、扩展和重用都带来不便。而观察者模式的关键对象是主题和观察者，一个主题可以有任意数目的观察者，一旦主题的状态发生了变更，所有的观察者都可以得到通知。主题发出通知时并不需要知道谁是它的观察者，也就是说，具体观察者是谁，它根本不需要知道。而任何一个具体观察者不知道也不需要知道其他观察者的存在。

观察者模式主要的工作其实就是在解除耦合。让耦合的双方都依赖于抽象，而不是依赖于具体。从而使得各自的变化都不会影响另一边的变化。

委托就是一种引用方法的类型。一旦为委托分配了方法，委托将于改方法具有完全相同的行为。委托方法的使用可以像其他任何方法一样，具有参数和返回值。委托可以看做是对函数的抽象，是函数的“类”。委托的实例将代表一个具体的函数。

使用场景：有多个子类共有的方法，且逻辑相同

案例地址：https://github.com/ben201708/learn\_java/tree/master/src/com/company/design\_mode/observer

**享元模式**

享元模式（Flyweight）,运用共享技术有效地支持大量细粒度的对象。

享元模式可以避免大量非常相似的类的开销。在程序设计中，有时需要生成大量细粒度的类的实例来表示数据。如果能发现这些实例除了几个参数外基本上都是相同的，有时就能够受大幅度地减少需要实例化的类的数量。如果能把那些参数移到类实例外面，在方法调用时将它们传递进来，就可以通过共享大幅度地减少单个实例的数目。

如果一个应用程序使用大量的对象，而大量的这些对象造成很大的存储开销，就可以考虑使用享元模式；还有就是对象的大多数状态可以外部状态，如果删除对象的外部状态，那么可以用相对较少的共享对象取代很多组对象，此时可以考虑使用享元模式。

使用场景：大量相似对象运用，需要缓冲池的场景

Integer a = 2;(-127~128之间采用缓存技术，可以使用==比较)

案例地址：https://github.com/ben201708/learn\_java/tree/master/src/com/company/design\_mode/flyweight

**迭代器模式**

迭代器模式(Iterator),提供一種方法顺序访问一个聚合对象中的各个元素，而又不报了该对象的内部表示。

当你需要访问一个聚集对象，而且不管对象是什么都需要遍历的时候，你就应该考虑用迭代器模式。为遍历不同的聚集结构提供如开始、下一个、是否结束、当前那一项等统一的接口。

案例：foreach in

**建造者模式**

建造者模式是设计模式的一种，将一个复杂对象的构建与它的表示分离，使得同样的构建过程可以创建不同的表示。

**建造者模式通常包括下面几个角色**：

1. **builder**：给出一个抽象接口，以规范产品对象的各个组成成分的建造。这个接口规定要实现复杂对象的哪些部分的创建，并不涉及具体的对象部件的创建。

2. **ConcreteBuilder**：实现Builder接口，针对不同的商业逻辑，具体化复杂对象的各部分的创建。 在建造过程完成后，提供产品的实例。

3. **Director**：调用具体建造者来创建复杂对象的各个部分，在指导者中不涉及具体产品的信息，只负责保证对象各部分完整创建或按某种顺序创建。

4. **Product**：要创建的复杂对象。

使用场景：okhttp创建

**模板方法模式**

在模板模式（Template Pattern）中，一个抽象类公开定义了执行它的方法的方式/模板。它的子类可以按需要重写方法实现，但调用将以抽象类中定义的方式进行。这种类型的设计模式属于行为型模式。

**原型模式**

用原型实例指定创建对象的种类，并且通过拷贝这些原型创建新的对象。

**外观模式**

外观模式（façade），为子系统中的一组接口提供一个一致的界面，定义一个高层接口，这个接口使得这一子系统更加容易使用。

**状态模式**

在状态模式（State Pattern）中，类的行为是基于它的状态改变的。这种类型的设计模式属于行为型模式。

在状态模式中，我们创建表示各种状态的对象和一个行为随着状态对象改变而改变的 context 对象。

状态模式主要解决的是当控制一个对象状态的条件表达式过于复杂时的情况。把状态的判断逻辑转移到表示不同状态的一系列类中，可以把复杂的判断逻辑简化。

**备忘录模式**

备忘录模式：在不破坏封闭的前提下，捕获一个对象的内部状态，并在该对象之外保存这个状态。这样以后就可将该对象恢复到原先保存的状态。

**桥接模式**

桥接模式是将抽象部分与它的实现部分分离，使它们都可以独立地变化。它是一种对象结构型模式，又称为柄体(Handle and Body)模式或接口(Interface)模式。

**命令模式**

命令模式（Command Pattern）是一种数据驱动的设计模式，它属于行为型模式。请求以命令的形式包裹在对象中，并传给调用对象。调用对象寻找可以处理该命令的合适的对象，并把该命令传给相应的对象，该对象执行命令。

**职责链模式**

责任链模式：在责任链模式里，很多对象由每一个对象对其下家的引用而连接起来形成一条链。请求在这个链上传递，直到链上的某一个对象决定处理此请求。发出这个请求的客户端并不知道链上的哪一个对象最终处理这个请求，这使得系统可以在不影响客户端的情况下动态地重新组织和分配责任。

**中介者模式**

中介者模式（Mediator Pattern）是用来降低多个对象和类之间的通信复杂性。这种模式提供了一个中介类，该类通常处理不同类之间的通信，并支持松耦合，使代码易于维护。中介者模式属于行为型模式。

**解释器模式**

解释器模式（Interpreter Pattern）提供了评估语言的语法或表达式的方式，它属于行为型模式。这种模式实现了一个表达式接口，该接口解释一个特定的上下文。这种模式被用在 SQL 解析、符号处理引擎等。

**访问者模式**

在访问者模式（Visitor Pattern）中，我们使用了一个访问者类，它改变了元素类的执行算法。通过这种方式，元素的执行算法可以随着访问者改变而改变。这种类型的设计模式属于行为型模式。根据模式，元素对象已接受访问者对象，这样访问者对象就可以处理元素对象上的操作。

**附件：**

案例代码编写：<https://github.com/ben201708/learn_java/tree/master/src/com/company/design_mode>

参考书籍：《大话设计模式》

设计模式参考文档：<http://www.runoob.com/design-pattern/command-pattern.html>