转载请注明连接：https://blog.csdn.net/feather_wch/article/details/78674418

> 十二个设计模式的73个知识点，按顺序复习和清理，对于梳理设计模式知识点很有帮助。
> 知识点源于《Head First 设计模式》

设计模式百题大全

版本：2018/9/2-1

1. 模式-的定义？

> 模式是在某情境(context)中，针对某些问题的某种解决方案。

2. 模式的分类：

> 1. 创建型、结构型、行为型。
> 2. 类型、对象型。

3. 本书所有模式和描述配对

模式	描述
装饰者	包装一个对象，以提供一个新的行为
模板方法	由子类决定如何实现一个算法中的步骤
抽象工厂	允许客户创建对象的家族，而无需指定他们的具体类

模式	描述
单件	确保有且只有一个对象被创建
策略	封装可以互换的行为，并使用委托来决定要使用哪一个
组合	客户用一致的方法来处理对象集合和单个对象
状态	封装了基于状态的行为，并使用委托在行为之间切换
迭代器	在对象的集合中游走，而不暴露集合的实现
外观	简化一群类的接口
装饰者	包装一个对象，以提供新的行为
工厂方法	由子类决定要创建的具体类是哪一个
观察者	让对象能在状态改变时被通知
代理	包装对象，以控制对该对象的访问
适配器	包装一个对象，并给该对象提供一个不同的接口
命令	将请求封装成对象

1-策略模式

1. 定义：

> 封装一系列算法族(行为族), 让他们之间可以相互替换。此模式使得算法族的改变独立于使用该算法族的客户。

2. 如何应用：

> 定义一个算法族/行为接口，用行为子类去实现接口，并在具体客户类去组合这些行为子类即可。

2. 设计原则一：封装变换部分

> 找出可能变换的部分，将其独立，使得某些部分的变化不会影响到其他部分。将这些变换的部分(例如行为)分离出来，组成一类。

3. 设计原则二：针对接口编程，而不是针对实现编程。

> 让行为类去实现行为接口，而不是用具体子类去实现行为接口。
> 关键在于利用了多态，使得可以在运行的时候根据实际情况去执行行为。

4. 设计原则三：多用组合，少用继承
5. 什么是组合?

> 例如策略章节中，将两个类结合起来使用，这就是组合。

6. 组合和继承的区别？以及继承的缺点

> 组合是在适当的时候通过与行为对象的组合，而产生了新的行为。而继承就是从父类继承而来。
> 继承的缺点在于：牵一发而动全身，而且难以在运行时改变自身的行为。

6. 组合的优点

> 可复用、可扩展、可维护

2-观察者模式

1. 定义

> 定义了对象之间的一对多依赖，当一个对象改变状态时，它的所有依赖者都会收到通知并且自动更新。(观察者模式是JDK中使用最多的模式)。

2. 设计原则四：交互对象之间松耦合设计
3. 如何使用?

> 具有subject接口和observer接口。subject中包含regitser和notification方法。用于在具体子类中(被观察者)实现注册观察者和通知观察者的方法。Observer接口中包含update更新方法，用于在子类观察者中实现如何更新的逻辑。使用时，在subject子类中register（注册）观察者observer并调用subject子类中的notification就能够实现通知流程。

4. Java内置Observable(被观察者)缺点

> - Observable是一个类而不是interface，影响了复用和使用。
> - 违反了多用组合，少用继承的原则。

5. 观察者模式是如何利用设计原则的?

> 1. 针对接口编程
>   Observer观察者利用主题的接口向Subject进行注册。
>   Subject使用Observer的接口通知观察者
> 2. 多组合少继承
>   对象之间的关系不是通过继承产生，而是在运行时通过组合来产生。

3-装饰者模式

1. 定义

> 动态的将责任附加到对象身上。若要扩展功能，装饰者提供了比继承更有弹性的替代方案。

2. 设计原则五：类应该对扩展开放，对修改关闭。
3. 实现中使用extends继承，不是应该避免继承吗

> 装饰者使用继承实现的，然而思想和继承不同。装饰者(Decorator)继承自组件(Component)，这里实际意义是“装饰”，原因在于，通过继承只是为了“类型匹配”而不是为了继承获得某些行为。

4. 装饰者如何获得新行为？(利用了组合)

> 在装饰的过程中，添加新的行为，这些行为不是继承于超类，这就是组合。

5. 实现方法：

> 1.定义一个抽象组件Component—为所有类的超类
> 2.定义抽象装饰者Decorator，继承自抽象组件C
> 3.实现各种具体组件，都继承自抽象组件C
> 4.实现各种具体装饰者，都继承自抽象装饰者D

6. Java中的装饰者

> Java中IO就是利用装饰者，TnputStream等等层层包裹

7. 装饰者的缺点：

> 设计中包含大量的小类，导致使用该API程序员的困扰。

4-工厂模式

1. 定义：

> 定义了一个创建对象的接口，单由子类决定要实例化的类是哪一个。工厂方法让类把实例化推迟到子类。

2. 依赖倒置原则(Dependency Inversion Principle)

> 要依赖抽象，不要依赖具体类

3. 遵循原则的指导方针

> - 变量不可以持有具体类的引用：使用new就会持有具体类的引用，这就可以使用Factory
> - 不要让类派生自具体类：会导致依赖具体类，应该派生自抽象(抽象类和接口)
> - 不要覆盖基类中实现的方法：如果需要覆盖就不是适合被继承的基类

4. 抽象工厂模式

> 提供了一个接口，用于创建相关或依赖对象的家族，而不需要明确指定具体类。

5. 工厂方法和抽象工厂的异同

> 1. 功能相同
>   都是负责将客户从具体类型中 解耦
> 2. 方法不同
>   工厂方法使用“继承”，抽象工厂利用“组合”
> 3. 方法的具体不同
>
> - 工厂方法通过子类创建对象，客户只需要知道所需的抽象类型，由子类负责决定具体类型。
> - 提供用来创建一个产品家族的抽象类型，该类型子类定义了产品产生的方法。使用该工厂，需要先实例化该工厂，再将其传入针对抽象类型所写的代码中。

5-单件模式

1. 定义

> 确保一个类只有一个实例，并提供一个全局访问点。

2. 有什么用?

> 适用于对象只要一个的情况，比如：线程池thread pool，cache，对话框，处理偏好设置，register(注册表)对象，日志对象，充当打印机、显卡等设备的驱动程序的对象。

3. 优点

> 单件模式提供一个全局访问点，和全局变量一样方便，却没有全局变量的缺点：需要一开始创建，假如其很消耗资源，却一直用不到，就会造成性能损耗。

4. 多线程

> 三种方案解决多线程：1.单纯同步 2.立即实例化 3.双重上锁

6-命令模式

1. 定义

> 将“请求”封装成对象，以便使用不同的请求、队列或者日志来参数化其他对象。命令模式也支持可撤销操作。

2. 结构

> 命令、接受命令执行者、被命令对象、客户

3. 撤销和宏命令

> 撤销操作可以使用栈记录执行过的命令，在undo时以此取出栈顶命令，并且执行其undo方法。
> 宏命令就是用数组记录所有命令，并且遍历执行。

4. 命令的更多用途

> 1、队列请求
> 2、日志请求

7-适配器模式、外观模式

1. 适配器模式定义

> Adapter将一个类的接口转换为另一个客户需要的接口，这样能使得原本接口不兼容的类能够一起工作。

2. OO适配器作用

> 将一个接口转换为客户需要的接口。使得客户和供应商各自的代码都不需要变换，只要一个中间作用的适配器即可。

3. 如何遵循原则

> 提供适配器类，将所有的改变封装在一个类中。
> 用可改变的接口将被适配者包裹起来---这样好处就是，任何被适配者子类(subclass)都可以配合adapter使用。

4. 两种适配器

> 1. object适配器
>   使用组合composition实现，上文实现的就是对象适配器。
> 2. class适配器
>   类适配器需要多重继承(multiple inheritance)。adapter是Target和Adaptee的子类，因此这在java中是无法实现的。所以本文不过多介绍。

5. 适配器在Java中的使用

> Enumerators和Iterators的适配

6. Adapter适配器和Decorator装饰者的区别

> 1.装饰者致力于给对象加上新的职责和行为。
> 2.当你需要一系列class一起工作，并且给客户提供需要的接口。这就需要适配器。

7. 外观模式作用

> 作用：让一个接口更简单。
> 使用组合实现外观模式。

8. 外观模式定义

> 外观模式给子系统的一系列接口提供了一个统一的接口。外观模式定义了一个高层接口，让子系统更容易使用。

9. 外观模式优点和缺点

> 1、能提供全部的功能(能直接操纵底层的功能)，也提供了一个简化的接口。
> 2、将用户实现和底层系统解耦。
> 3、解耦，减少系统维护成本
> 4、缺点：创造了更多的包装类，会增加复杂度，开发时间和运行性能。

10. Facade和Adapter的区别

> Adapter的目的是将一个接口转换为用户需要的接口。Facade的目的是提供一个简化的接口。

11. 设计原则六

> 最少知识原则：只和你的密友谈话-要减少对象之间的交互，只留下几个“密友”。
> Java中System.out.println就违反了该原则：所有的原则都需要在实际的情况中去选择性的采用。

12. 如何不影响太多的对象

> 只调用下列范畴的方法。
>
> 1. 该对象本身，的方法
> 2. 作为参数传入方法的对象，的方法
> 3. 该方法创建的任何对象，的方法
> 4. 该对象任何组件，的方法
>
> 如果某对象是调用其他对象返回的结果，不要调用该对象的方法。

8-模板方法模式

1. 定义和作用

> 模板方法在method中定义了算法的骨架并将一些步骤推迟到子类中实现。该模式在不改变算法结构的情况下，让子类重新定义了算法中某些步骤的内容。

2. 使用场景
3. 设计模式：The Hollywood Principle

> **Don't call us，we'll call you.**
> 通过该原则，允许底层的组件可以将自己挂载到系统中，而不是高层组件决定何时用它们。

4. hook的作用

> 子类可以重写该方法添加相应的判断，也可以不重写，这就是hook的用途。

5. 好莱坞原则和依赖倒置原则区别

> 都是致力于解耦(decouple)
>
> 1. 依赖倒置原则：是致力于避免使用具体类，而更多的使用抽象
> 2. 好莱坞原则：是致力于建立一种框架或组件，以允许底层组件能挂载到系统中，而不用创建底层和高层之间的依赖。

6. Java API中模板方法的使用

> 比如数组排序方法 `sort` 里面只调用了 `mergeSort(...)` ，mergeSort就可以看做模板方法。里面定义了排序的步骤，使用了compareTo、swap等步骤。

7. 模板方法和策略模式的区别

> - ◦ Template Method控制算法的框架，而把具体的有区别的步骤交给子类实现。
> - 策略模式放弃控制算法。致力于给用户提供多组可选择的算法。用组合的方式实现。
> - ◦ Template Method有更少的对象，更有效率(一点点)。
> - 策略模式使用组合而更加灵活，而且可以在运行时更改算法。
> - ◦ Template Method依赖父类中实现的方法。
> - 策略模式不依赖任何人，甚至能自己实现一整套算法。

9-Iterator和Composite模式

1. 迭代器定义和作用

> 提供一种方法能够顺序访问聚合对象中的元素，而不需要暴露内部的表示。迭代器遍历一个聚合物，并将其封装成另一个对象。

2. 迭代器适用场景
3. 迭代器模式的结构和实现方法

> 1. 聚合接口(Aggregate)-提供createIterator方法
> 2. ConcreteAggregate-实现聚合接口中的createIterator方法
> 3. Iterator(迭代器接口)-提供hasNext\next等方法
> 4. ConcreteIterator(具体迭代器)-实现接口的方法

4. 组合模式定义和作用以及适用场景

> 允许你将对象组合成树形结构来构成“整体/部分”的层次结构。组合能让客户一致地处理单一对象和对象组合。
> 适用于“整体/部分”的关系结构，用于统一处理所有对象(不分节点还是叶子)。

5. 组合模式结构和实现的两种方案。

> - 类似于树的结构，有节点和叶子节点。但是系统在出的时候把叶子节点看做没有分支的节点。
>
> 1. Compoent(组件)-包含叶子节点和节点的所有操作。
> 2. Composite(组合，节点，extends Component)-实现属于节点的操作
> 3. Leaf(叶子节点，extends Component)-实现叶子节点的操作
> 4. 将叶子节点和节点组成最终的ALL节点。形成一棵树，而ALL节点就是树根。
> 5. 对树根进行操作，会将Leaf和Composite都当做Compoent一起处理。
>
> - 将所有功能都加入到Component中。或者将不同功能通过接口来赋予不同对象。其间的思想就是透明度和安全性的平衡。

6. 迭代器设计原则：一个类一个功能。
7. 观察者和状态模式的区别

> - 观察者：当状态改变时，通知一组对象。
> - 状态：当状态改变时，让一个对象改变行为。

8. 组合模式优点

> 能够让客户可以处理多个对象，而不需要知道对象实际的类型。

10-状态模式

1. 定义

> 允许一个对象在内部状态改变时改变自己的行为。这个对象就像是改变了他的类。

2. 结构

> 1. State(interface)状态接口
> 2. Concrete State：具体状态
> 3. Context：实例中的GumballMachine，持有各种状态的实例。
> 4. context中调用state.handle()：进行处理，在不同状态下做不同工作。

3. 实现方法和应用场景
4. State状态模式和Strategy策略模式的异同

> 1. 相同
>   结构类似
> 2. 不同
>
> - State状态模式是一个对象的状态改变后，会改变自身的行为
> - Strategy是给一个对象封装好的算法族(行为)，可以更改算法族(行为)

11-代理模式

1. 定义

> 代理模式为另一个对象提供一个替身用于控制对这个对象的访问。
>
> 使用代理模式创建代表，让代表对象控制某对象的访问，被代理的对象可以是远程对象，创建开销大的对象或者需要安全控制的对象。

2. 结构

> 1. Subject接口：该接口允许任何客户都可以像处理RealSubject对象一样去处理Proxy对象。
> 2. Proxy(实现Subject)：持有RealSubject对象的引用，需要时将请求转发给RealSubject
> 3. RealSubject(实现Subject)：真正对象，该对象的访问会被Proxy对象控制。

3. RMI/虚拟代理/动态代理是什么?

> - RMI是java RMI框架 远程方法调用。
> - Java在java.lang.reflect包中提供代理支持。可以动态的创建代理类。这就是动态代理。更多代理模式查阅资料学习吧。

4. 代理模式和装饰者模式的区别

> 代理模式：包装另一个对象，并控制对它的访问。
> 装饰者模式：包装另一个对象，并添加额外的行为。

12-复合模式

1. MVC的组成

> M：model模型，持有所有的数据、状态和程序逻辑。提供了操作和解锁状态的接口，并发送状态改变给观察者。
> V：View视图，从模型中获取需要显示的状态和数据。
> C：Control控制层，获取用户输入，并且解读其对model的意思。

2. MVC的内在设计模式

> - Model让View和Control可以随状态改变而更新，是使用**观察者模式**
> - View和Control实现了策略模式：Ctrl是View的行为，如果要不同行为，可以换掉C层。
> - View内部用组合模式来管理窗口、按钮以及其他显示组件。