使用设计模式的目的是为了可重用代码，提高代码的可扩展性和可维护性。

开闭原则：扩展开放，修改关闭。

里氏代换原则：使用父类的地方都能使用子类对象

# 创建型模式

创建型模式关注点是如何创建对象，其核心思想是要把对象的创建和使用相分离。

创建型模式包括：

* 工厂方法：Factory Method
* 抽象工厂：Abstract Factory
* 建造者：Builder
* 原型：Prototype
* 单例：Singleton

## 工厂方法

Integer n = Integer.valueOf(100);

Integer既静态工厂。它提供了静态方法valueOf()来创建Integer。那么这种方式和直接写new Integer(100)有何区别呢？我们观察valueOf()方法：

1. **public** **final** **class** Integer { **public** **static** Integer valueOf(**int** i) {
2. **if** (i >= IntegerCache.low && i <= IntegerCache.high) **return** IntegerCache.cache[i + (-IntegerCache.low)];
3. **return** **new** Integer(i); } ... }

它的好处在于，valueOf()内部可能会使用new创建一个新的Integer实例，但也可能直接返回一个缓存的Integer实例。对于调用方来说，没必要知道Integer创建的细节。

 工厂方法可以隐藏创建产品的细节，且不一定每次都会真正创建产品，完全可以返回缓存的产品，从而提升速度并减少内存消耗。

## 抽象工厂

抽象工厂模式是为了让创建工厂和一组产品与使用相分离，并可以随时切换到另一个工厂以及另一组产品；

抽象工厂模式实现的关键点是定义工厂接口和产品接口，但如何实现工厂与产品本身需要留给具体的子类实现，客户端只和抽象工厂与抽象产品打交道

## 建造者：Builder

生成器模式（Builder）是使用多个“小型”工厂来最终创建出一个完整对象。

## 原型：Prototype

是指创建新对象的时候，根据现有的一个原型来创建。 复制过来

原型模式应用不是很广泛，因为很多实例会持有类似文件、Socket这样的资源，而这些资源是无法复制给另一个对象共享的，只有存储简单类型的“值”对象可以复制。

## 单例：Singleton

单例模式（Singleton）的目的是为了保证在一个进程中，某个类有且仅有一个实例。

public class Singleton { // 静态字段引用唯一实例: private static final Singleton INSTANCE = new Singleton(); // 通过静态方法返回实例: public static Singleton getInstance() { return INSTANCE; } // private构造方法保证外部无法实例化: private Singleton() { } }

那我们什么时候应该用Singleton呢？实际上，很多程序，尤其是Web程序，大部分服务类都应该被视作Singleton，如果全部按Singleton的写法写，会非常麻烦，所以，通常是通过约定让框架（例如Spring）来实例化这些类，保证只有一个实例，调用方自觉通过框架获取实例而不是new操作符：

@Component // 表示一个单例组件 public class MyService { ... }

# 结构型模式

结构型模式主要涉及如何组合各种对象以便获得更好、更灵活的结构。

* 适配器
* 桥接
* 组合
* 装饰器
* 外观
* 享元
* 代理

## 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，使得原本由于接口不兼容而不能一起工作的那些类可以一起工作*

Arrays.asList(T[])就相当于一个转换器，它可以把数组转换为List

类似还有InputStreamReader就是Java标准库提供的Adapter，它负责把一个InputStream适配为Reader

客户端通过自己选择一个品牌，再配合一种引擎，得到最终的Car：

RefinedCar car = **new** BossCar(**new** HybridEngine());

car.drive();

## 桥接

*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*

客户端通过自己选择一个品牌，再配合一种引擎，得到最终的Car：

RefinedCar car = **new** BossCar(**new** HybridEngine());

car.drive();

使用桥接模式的好处在于，如果要增加一种引擎，只需要针对Engine派生一个新的子类，如果要增加一个品牌，只需要针对RefinedCar派生一个子类，任何RefinedCar的子类都可以和任何一种Engine自由组合,即一辆汽车的两个维度：品牌和引擎都可以独立地变化

## 组合Composite

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

*如*XML或HTML的实现

## 装饰器Decorator

是一种在运行期动态给某个对象的实例增加功能的方法 感觉像代理

## 外观Facade

如果客户端要跟许多子系统打交道，那么客户端需要了解各个子系统的接口，比较麻烦。如果有一个统一的“中介”，让客户端只跟中介打交道，中介再去跟各个子系统打交道，对客户端来说就比较简单。所以Facade就相当于搞了一个中介

## 享元Flyweight

如果一个对象实例一经创建就不可变，那么反复创建相同的实例就没有必要，直接向调用方返回一个共享的实例就行，这样即节省内存，又可以减少创建对象的过程，提高运行速度

Integer为例，如果我们通过Integer.valueOf()这个静态工厂方法创建Integer实例，当传入的int范围在-128~+127之间时，会直接返回缓存的Integer实例 从而实现不可变实例的复用

## 代理

### 远程代理

### 虚代理

虚代理即Virtual Proxy，它让调用者先持有一个代理对象，但真正的对象尚未创建。如果没有必要，这个真正的对象是不会被创建的，直到客户端需要真的必须调用时，才创建真正的对象。JDBC的连接池返回的JDBC连接（Connection对象）就可以是一个虚代理，即获取连接时根本没有任何实际的数据库连接，直到第一次执行JDBC查询或更新操作时，才真正创建实际的JDBC连接。

### 保护代理

### 智能引用

# 行为型模式

行为型模式主要涉及算法和对象间的职责分配

* 责任链
* 命令
* 解释器
* 迭代器
* 中介
* 备忘录
* 观察者
* 状态
* 策略
* 模板方法
* 访问者

## 责任链

责任链模式是一种把多个处理器组合在一起，依次处理请求的模式；

责任链模式经常用在拦截、预处理请求等。

## 命令

命令模式的设计思想是把命令的创建和执行分离，使得调用者无需关心具体的执行过程。

## 解释器

解释器模式通过抽象语法树实现对用户输入的解释执行。

类似的，当我们使用JDBC时，执行的SQL语句虽然是字符串，但最终需要数据库服务器的SQL解释器来把SQL“翻译”成数据库服务器能执行的代码，这个执行引擎也非常复杂，但对于使用者来说，仅仅需要写出SQL字符串即可。

## 迭代器

因为Iterator模式十分有用，因此，Java允许我们直接把任何支持Iterator的集合对象用foreach循环写出来：

**List**<String> **list** = ...

**for** (String s : **list**) {

}

## 中介

中介模式是通过引入一个中介对象，把多边关系变成多个双边关系，从而简化系统组件的交互耦合度。

Mediator模式经常用在有众多交互组件的UI上。为了简化UI程序，MVC模式以及MVVM模式都可以看作是Mediator模式的扩展。

## 备忘录

其实我们使用的几乎所有软件都用到了备忘录模式。最简单的备忘录模式就是保存到文件，打开文件。对于文本编辑器来说，保存就是把TextEditor类的字符串存储到文件，打开就是恢复TextEditor类的状态。对于图像编辑器来说，原理是一样的，只是保存和恢复的数据格式比较复杂而已。Java的序列化也可以看作是备忘录模式。

## 观察者

观察者模式，又称发布-订阅模式，是一种一对多的通知机制，使得双方无需关心对方，只关心通知本身。

## 状态

状态模式的设计思想是把不同状态的逻辑分离到不同的状态类中，从而使得增加新状态更容易；

简单ai案例很有意思

https://www.liaoxuefeng.com/wiki/1252599548343744/1281319592001569

## 策略

策略模式是为了允许调用方选择一个算法，从而通过不同策略实现不同的计算结果

## 模板方法

模板方法是一种高层定义骨架，底层实现细节的设计模式，适用于流程固定，但某些步骤不确定或可替换的情况

## 访问者

访问者模式是为了抽象出作用于一组复杂对象的操作，并且后续可以新增操作而不必对现有的对象结构做任何改动