常用的设计模式

1. 创建型
    - **Factory Method (工厂方法)**：“**让子类决定生产什么对象**”，父类定规则，子类做决定，把对象的创建延迟到子类，避免硬编码 new
        - 解决了什么问题?
            - 解耦，客户端（`Main` 类）不需要知道具体实现，只依赖抽象
            - 扩展性，新增种类时只需要增加新的实现类
            - 符合开闭原则
    - **Abstract Factory (抽象工厂方法)**：“**一站式生产整套产品**”，定义一个超级工厂接口，让具体工厂生产同一品牌的全套产品
        - 解决了什么问题?
            - 保证产品兼容性
            - 切换产品族方便
            - 符合开闭原则
    - **Builder(建造者模式)**：“**分步骤组装复杂对象**”，当一个对象参数很多，用构造函数传参会很臃肿，并且难以扩展。可以采用 Builder 模式，**分步骤设置参数**，最后统一组装对象
        - 解决了什么问题?
            - 参数灵活，可以只设置需要的参数
            - 代码可读，链式调用清晰表达组装逻辑
            - 避免无效状态，通过 `build()` 方法统一校验参数
    - **Singleton(单例模式)**：“**一个类只能有一个实例**”，某些对象（如配置管理器、数据库连接池）必须**全局唯一**，重复创建会浪费资源或导致数据不一致
2. 结构型
    - **Adapter(适配器模式)**：“**转换接口，兼容新旧**”，两个系统或类因为**接口不兼容**无法直接合作（比如老代码调用新库，或第三方接口不匹配）。适配器模式加一个 **中间层（适配器）**，把 “旧接口” 转换成 “新接口”
        - 举个例子，假设你的电商系统需要同时支持 **微信支付** 和 **支付宝**，但它们的接口完全不同：微信的接口返回 boolean 表示成功与否，支付宝返回 PayResult 对象。但你希望对外提供统一的支付接口，屏蔽底层差异，这里就可以使用 adaptor
    - **Bridge(桥接模式)**：“**把抽象和实现分开，让它们自由组合**”，当一个类有多个变化维度（比如“形状”和“颜色”），用继承会导致类爆炸（不用为每种组合创建子类）
        - 类比一下，**抽象** 相当于遥控器的按钮（开/关、调音量），**实现** 相当于不同品牌的电视（索尼、小米），**桥接** 相当于遥控器持有电视的引用，但不知道具体品牌
    - **Decorator(装饰器模式)**：“**给对象动态加功能，像套娃一样层层包装**”，当我们想给一个对象添加新功能，但不想修改它的源代码（避免破坏原有逻辑），就可以用 **装饰器类** 包裹原始对象，在 **不改变原对象** 的基础上，动态添加功能
    - **Flyweight(享元模式)**：“**共享相同部分，节省内存**”，当程序需要创建**大量相似对象**时（比如游戏中的子弹、棋子、文字），每个对象都独立存储数据会浪费内存，我们就可以采用 **享元模式**，拆分对象的属性为内部状态（不变的、可共享的数据）和外部状态（变化的，不可共享的数据）

- **Proxy(代理模式)**：“**找个替身帮你干活**”，直接访问某个对象可能会带来问题（比如性能开销、权限控制、远程调用等），于是我们引入一个代理对象，在客户端和目标对象之间加一层控制，由代理决定如何处理请求

3. 行为型

- **Template(模板方法模式)**：“**定好流程大纲，具体步骤自己填**”，多个任务有 **相同的流程**，但某些步骤的具体实现不同（比如做咖啡和泡茶，步骤类似，但细节不同），于是我们可以抽象父类的整体流程（模板方法），子类只需要实现自己需要定制的步骤
- **Method(方法模式)**：“**把复杂操作拆成小步骤，让代码更清晰**”，当一个方法过于复杂（比如几十行代码，混合了多种逻辑），会导致难以阅读和维护，并且无法复用部分代码，于是我们**把大方法拆分成多个小方法**，每个小方法只做一件事，再组合起来完成大功能
- **Chain Of Responsibility(责任链模式)**："**踢皮球，一个请求在多个对象之间传递**"，一个请求需要经过多个处理者，但不确定具体由谁处理（比如不同金额的报销需要不同级别审批），于是我们把多个处理者串成一条链，请求沿着链传递，直到被处理或链结束
  - 解决了什么问题？
    - 解耦，请求发送者不需要知道具体谁处理
    - 动态调整，可以随时增减或调整处理者顺序
    - 灵活性，每个处理者只需关注自己的责任范围
- **Iterator(迭代器模式)**：“**统一遍历所有对象，不关心底层结构**”，不同的集合（如数组、链表、树）遍历方式不同，直接暴露内部结构会让代码混乱，于是提供统一的遍历接口，隐藏集合的内部实现
  - 解决了什么问题？
    - 解耦，遍历代码不依赖具体集合实现（数组、链表、树都能用同一套遍历逻辑）
    - 简化调用，用户只需调 `hasNext()` 和 `next()`，不用关心底层是 `for` 循环还是 `while`
- **Observer(观察者模式)**：“**微信订阅号，作者发布，粉丝自动收到推送**”，当一个对象（如天气数据）变化时，需要自动通知多个依赖对象（如手机App、广告牌），但 **不想让它们紧密耦合**，我们可以这样解决：
  - **被观察者（Subject）**：维护一个订阅者列表，提供 `添加/删除` 订阅的方法
  - **观察者（Observer）**：定义统一的更新接口（如 `update()`）
  - **流程**：被观察者状态变化 → 遍历订阅者列表 → 调用每个观察者的 `update()`
- **State(状态模式)**：“**对象变，行为跟着变**”，当一个对象的行为 **取决于它的状态**（比如订单的“待支付”、“已发货”），如果用一堆 `if-else` 判断状态，代码会臃肿且难维护，于是我们可以**把状态抽成独立的类**，对象内部切换状态，行为自动变化
- **Strategy(策略模式)**：“**算法随便换，调用不改变**”，当一个任务有 **多种算法**（如排序、支付、导航），用 `if-else` 硬编码会导致代码臃肿且难扩展，于是我们可以定义算法接口，将每种算法封装成独立类，运行时动态的切换算法，调用方无需关心细节
  - 比如导航 App，选“最快路线”或“最省钱路线”，App 自动计算，但导航按钮始终是同一个；支付系统，用户选微信支付或支付宝，支付流程不变，只是底层实现不同
  - 解决了什么问题？
    - 开闭原则，新增算法只需要加新类，不需要改旧代码
    - 复用性，同一套算法可被多个场景共享

面向对象（OO）的设计法则

1. 单一职责原则：一个类只干一件事
2. 开闭原则：对扩展开放，对修改关闭
3. 里氏替换原则：子类必须能替换父类，且行为一致（防止调用父类方法时，子类抛出 **不支持的操作** 异常）
4. 接口隔离原则：接口要小而专，不能让用户实现用不到的方法
    - 比如：Animal 接口有 fly()、swim()、run()，但 Dog 类被迫实现无用的 fly()
5. 依赖倒置原则：依赖抽象，不依赖具体
    - 比如：
        - 错误设计：OrderService 直接调用 MySQLDatabase
        - 正确设计：OrderService 依赖 Database 接口，具体用 MySQL 还是 Mongo 由配置决定
6. 迪米特法则：少管闲事，只和朋友通信（减少类之间的耦合）
7. 组合由于继承原则：多用组合，少用继承