设计模式

- 设计原则
  - 单一职责原则
    - 定义
      - 应该有且仅有一个原因引起类的变更。也就是一个接口或类只有一个职责，它就负责一件事情。(PS：比如一个类中不可能同时存在修改，添加和删除某一个属性，因此这之中只能存在一个，这一个就是那个引起类变更的原因)
    - 好处
      - 类的复杂性降低，实现什么职责都有清晰明确的定义
      - 可读性提高，复杂性降低，那当然可读性提高了
      - 可维护性提高，可读性提高，那当然更容易维护
      - 变更引起的风向降低，变更时必不可少的，如果接口的单一职责做得好，一个接口修改只对应相应的实现类有影响，对其他的接口无影响，这对系统的扩展性、维护性都有非常大的帮助
    - 举例说明
      - 如pojo负责属性，service处理业务，DAO数据库操作，controller负责向页面传递数据等，不同的entity有着不同的使命，有userentity，classentity等
  - 里氏替换原则
    - 定义
      - 如果对每一个类型为S的对象o1，都有类型为T的对象o2，使得以T定义的所有程序P在所有的程序P在所有的对象o1都代换成o2时，程序P的行为没有发生变化，那么类型S是类型T的子类型
      - 所有引用基类的地方必须能透明地使用其子类的对象。通俗的讲就是只要父类能够出现的地方子类就可以出现，而且替换为子类也不会产生任何错误或异常，使用者可能根本不需要知道是父类还是子类。但是反过来就不行了，有子类出现的地方，父类未必就能适应
    - 含义
      - 子类必须完全实现父类的方法
      - 子类可以有自己的个性
      - 覆盖或实现父类的方法是输入参数可以被放大。子类中方法的前置条件必须与超类中的被覆写的方法的前置条件相同或是更宽松。比如：父类方法传入的参数是HashMap，子类方法传入的参数可以是Map
      - 覆写或实现父类的方法时输出结果可以被缩小
    - 这个主要是描述父子类
  - 依赖倒置原则
    - 含义

- 高层模块不应该依赖底层模块，两者都应该依赖其抽象（模块间的依赖通过抽象发生，实现类之间不发生直接的依赖关系，其依赖关系是通过接口或抽象类产生的）
- 抽象不应该依赖细节(接口或抽象类不依赖于实现类)
- 细节应该依赖抽象(实现类依赖接口或抽象类)

- 三种写法
  - 构造函数传递依赖对象
  - Setter方法传递依赖对象
  - 接口声明依赖对象
- 依赖正置就是类间的依赖是实实在在的实现类之间的依赖，也就是面向实现或过程编程

- 开闭原则
  - 定义
    - 一个软件实体如类、模块和函数应该对扩展开放(扩展类，复写方法)，对修改关闭(不要修改原有的方法)
  - 变化总结
    - 逻辑变化
      - 只变化一个逻辑，而不设计其它模块
    - 子模块变化
      - 一个模块变化，会对其它的模块产生影响，特别是一个底层次的模块变化必然引起高层模块的变化
    - 可见视图变化
      - 可见视图是提供给客户使用的界面
  - 重要性
    - 对测试的影响(指单元测试，不用修改以前的单元测试)
    - 提高复用性
    - 提高可维护性
    - 面向对象开发的要求
  - 使用原则
    - 抽象约束
      - 通过接口或抽象类约束扩展，对扩展进行边界限定，不允许出现在接口或抽象类中不存在的public方法
      - 参数类型、引用对象尽量使用接口或者抽象类，而不是实现类
      - 抽象层尽量保持稳定，一旦确定即不允许修改
    - 元数据控制模块行为
  - 这个主要是描述的在设计的时候需要考虑业务的扩展，而且在后期修改代码的时候尽量不要修改原有已经在运行的代码，考虑用扩展来实现
- 接口隔离原则
  - 定义
    - 客户端不应该依赖它不需要的接口
    - 类间的依赖关系应该建立在最小的接口上
  - 定义理解

- 建立单一接口，不要建立臃肿庞大的接口。再通俗点就是接口尽量细化，同时接口中的方法尽量少
- 含义
  - 接口要尽量小
  - 接口要高内聚
  - 定制服务
  - 接口设计是有限度的
- 衡量规则
  - 一个接口只服务于一个子模块或业务逻辑
  - 通过业务逻辑压缩接口中的public方法，接口时常去回顾，尽量让接口精干而不是一大堆方法
  - 已经被污染了的接口，尽量去改，若变更的风险较大，则采用适配器模式进行转化处理
  - 了解环境，拒绝盲从。每个项目或产品都有特定的环境因素，所以环境不同，接口拆分的标准就不同。
- 迪米特法则
  - 定义
    - 一个对象应该对其他对象有最少的了解，其核心就是类间解耦
  - 定义理解
    - 一个类应该对自己需要耦合或者调用的类知道得最少，你(被耦合或者调用的类)的内部是如何复杂都和我没有关系，那是你的事情，我就知道你提供的这么多public方法，我就调用这么多，其它的我一概不关心
  - 含义
    - 只和朋友交流
      - 只和有关的类进行耦合
    - 朋友之间是有距离的
      - 一个类公开的public属性或方法越多，修改时设计的面也就越大，变更引起的风险扩散也就越大。所以对于一个流程的所有方法应该用一个方法组合起来
    - 是自己的就是自己的
      - 如果一个方法放在本类中，既不增加类间的关系，也对本类不产生负面影响，那就放置在本类中
    - 谨慎使用Serializable
      - 注意客服端和服务器端是否同步更新
  - 衡量规则
    - 如果一个类跳转两次以上才能访问到另一个类，就需要想办法进行重构了，为什么是两次以上呢？因为一个系统的成功不仅仅是一个标准或是原则就能够决定的，有非常多的外在因素决定，跳转次数越多，系统越复杂，维护就越困难，所以只要跳转不超过两次都是可以忍受的，这需要具体问题具体分析。
- 创建型模型
  - 单例模式
    - 定义
      - 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

- 通用代码
- 优缺点
  - 优点
    - 由于在内存中只有一个实例，减少了内存开支，特别是一个对象需要频繁地创建、销毁时，而且创建或销毁时性能又无法优化的时候
    - 减少了系统性能开销。当一个对象的产生需要比较多的资源时，如读取配置、产生其他依赖对象时，则可以通过在应用启动时直接产生一个单例对象，然后用永久驻留内存的方式来解决。注意JVM的垃圾回收机制
    - 可以避免对资源的多重占用。如写文件动作
    - 可以在系统设置全局的访问点，优化和共享资源访问
  - 缺点
    - 一般没有接口，扩展很困难
    - 对测试不利。在并行开发环境中，如果单例模式没有完成，是不能进行测试的，没有接口也不能使用mock
    - 与单一职责原则有冲突。一个类应该只实现一个逻辑
- 使用场景
  - 要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
  - 创建一个对象需要消耗的资源过多，如IO和数据库
- 注意事项
  - 高并发情况下注意单例模式的线程同步问题
    - 饿汉式单例
    - 懒汉式单例(有synchronized)
  - 考虑对象的复制，一般是不需要复制的，即不实现Cloneable接口
- 理解
  - 无论怎么访问，都是最开始的那个类，而且有且仅有一个

- 原型模式
  - 定义
    - 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(不通过new关键字来产生一个对象，而是通过对象复制来实现的模式)
  - 通用代码
  - 优点
    - 性能优良
      - 原型模式是在内存二进制流的拷贝，要比直接new一个对象性能好很多，特别是要在一个循环体内产生大量的对象时
    - 逃避构造函数的约束
      - 直接在内存中拷贝，构造函数不会被执行
  - 使用场景
    - 资源优化
      - 类初始化需要消化非常多的资源，包括数据、硬件资源等

- 性能和安全要求的场景
  - 通过new产生一个对象需要非常繁琐的数据准备和访问权限，则可以使用原型模式
- 一个对象多个修改者的场景
  - 一个对象需要提供给其他对象访问，而且各个调用者可能都需要修改其值时，可以考虑使用原型模式拷贝多个对象供调用者使用
- 注意事项
  - 构造函数不会被执行
  - 浅拷贝和深拷贝
    - 浅拷贝(Cloneable) Java做了一个偷懒的拷贝动作，Object类提供的方法clone只是拷贝本对象，其对象内部的数组、引用对象等都不拷贝，还是指向原生对象的内部元素地址(使用原型模式时，引用的成员变量必须满足两个条件才不会被拷贝：一是类的成员变量，而不是方法内变量；二是必须是一个可变的引用对象，而不是一个原始类型或不可变对象。)
    - 深拷贝：拷贝对象之间互不影响，即在拷贝的时候将数组或对象一起拷贝
  - clone与final两个冤家
    - 对象的clone与对象内的final关键字是有冲突的(即要使用clone方法，类的成员变量上不要增加final关键字)
- 理解
  - clone出来的类

- 建造者模式
  - 定义
    - 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
  - 通用代码
  - 优点
    - 封装性
    - 建造者独立，容易扩容
    - 便于控制细节风险
  - 适用场景
    - 相同的方法，不同的执行顺序，产生不同的事件结果时
    - 多个部件或零件，都可以装配到一个对象中，但是产生的运行结果又不相同时
    - 产品类非常复杂，或者产品类中调用顺序不同产生了不同的效能
    - 在对象创建过程中会使用到系统中的一些其他对象，这些对象在产品对象的创建过程中不容易得到
  - 注意事项
    - 建造者模式关注的是零件类型和装配工艺(顺序)
  - 对比工厂模式
    - 建造者模式最主要的功能是基本方法的调用顺序安排，也就是这些基本方法已经实现了，通俗地说就是零件的装配，顺序不同产生的对象也不同；而工厂方法则重点是创建，创建零件是它的主要职责，组装顺序则不是它关心的。
  - 理解

- 重点在于基本方法的调用顺序，这个里面还有一个导演类(Director)，用于控制什么部件组装什么什么养得产品
- 工厂方法模式
  - 定义
    - 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
  - 通用代码
  - 优点
    - 良好的封装性，代码结构清晰
    - 扩展性好
    - 屏蔽产品类
    - 解耦框架。高层模块只需要知道产品的抽象类，其他实现类都不用关心
  - 使用场景
    - 是new一个对象的替代品，但是要考虑是否要增加一个工厂类进行管理，增加代码的复杂度
    - 需要灵活的、可扩展的架构时
    - 产品族内的约束为非公开状态
    - 异构项目中，就是和其它项目对接
  - 扩展
    - 简单工厂模式(静态工厂模式)
      - 去掉抽象类Creator，同时把createProduct设置为静态类型
      - 通用代码
    - 多个工厂类
      - 每个具体的产品类都对应了一个工厂类
      - 在复杂的应用中使用，然后再增加一个协调类，避免调用者与各个子工厂交流，协调类的作用是封装子工厂类，对高层模块提供统一的访问接口
      - 优缺点
        - 由于扩展一个产品类就需要建立一个相应的工厂类，因此增加了扩展的难度
        - 由于工厂类和产品类的数量相同，维护时需要考虑两个对象之间的关系
    - 替代单例模式
      - 通过反射方式创建
      - 再扩展一个单例构造器，输入一个类型就可以获得唯一的一个实例，但是类一定要有private的构造方法
    - 延迟初始化
      - 一个对象被消费完毕后，并不立刻释放，工厂类保持其初始状态，等待再次被使用(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)
      - 通用代码
- 抽象工厂模式
  - 定义
    - 为创建一组相关或相互依赖的对象提供接口，而且无须指定他们的具体关系(产品族)

- 通用代码
- 优缺点
  - 优点
    - 封装性好
  - 缺点
    - 产品族扩展困难，因为扩展将会改很多代码
- 使用场景
  - 一个对象族(或是一组没有任何关系的对象)都有相同的约束，则可以使用抽象工厂模式
  - 举例
    - 例如一个应用，需要在三个不同平台（Windows、Linux、Android（Google发布的智能终端操作系统））上运行。通过抽象工厂模式屏蔽掉操作系统对应用的影响。三个不同操作系统上的软件功能、应用逻辑、UI都应该是非常类似的，唯一不同的是调用不同的工厂方法，由不同的产品类去处理与操作系统交互的信息。
- 注意事项
  - 横向扩展容易，纵向扩展困难

- 设计模式对比
  - 工厂方法VS建造者
    - 工厂方法注重的是整体对象的创建
      - 注重点在整体
    - 建造者注重的是部件的构建构成，通过一步一步地精确构造创建出一个复杂的对象
      - 注重点在零件细节
  - 抽象工厂VS建造者
    - 抽象工厂实现对产品家族的创建；抽象工厂不需要关心构建过程，只关心什么产品由什么工厂生产即可
    - 建造者要求按照指定的蓝图建造产品，主要目的是通过组装零件而产生一个新产品
  - 代理VS装饰
    - 代理模式着重对代理过程的控制
    - 装饰模式是代理模式的一种特殊应用，但是它着重的是对类的功能进行加强或减弱，它着重类的功能变化
  - 装饰VS适配器
    - 相同点
      - 都是包装作用，都是通过委托方法实现其功能
    - 不同点
      - 装饰模式包装的是自己的兄弟类，属于同一个家族
      - 适配器模式则修饰非血缘关系的类，把一个非本家族的对象伪装成本家族的对象，注意是伪装，因此它的本质还是非相同接口的对象
  - 命令VS策略
    - 策略模式的意图是封装算法，让这些算法独立，兵器可以相互替换，让行为的变化独立于拥有行为的客户

- 命令模式对动作的解耦，把一个动作的执行分为执行对象、执行行为，让两种互相独立而不相互影响
- 策略VS状态
  - 策略模式封装的是不同的算法，算法之间没有交互，以达到算法可以切换自由的目的
  - 状态模式封装的是不同的状态，以达到状态切换行为随之发生变化的目的
- 策略VS桥梁
  - 策略模式旨在封装一系列的行为
  - 桥梁模式则是解决在不破坏封装的情况下如何抽取出它的抽象部分和实现部分，它的前提是不破坏封装，让抽象部分和实现部分都可以独立地变化
- 门面VS中介者
  - 门面模式为复杂的子系统提供一个统一的访问界面，它定义的是一个高层接口，该接口使得子系统更加容易使用，避免外部模块深入到子系统内部而产生与子系统内部细节耦合的问题
  - 中介者模式使用一个中介对象来封装一系列同事对象的交互行为，它使各对象之间不再显示地应用，从而使其耦合松散，建立一个可扩展的应用架构
- 工厂方法VS抽象工厂
  - 工厂方法产生的是一个产品
  - 抽象工厂产生的是一系列的产品族

- 结构型模型
  - 适配器模式
    - 定义(类适配器)
      - 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。换句话说，适配器模式就是把一个接口或类转换成其他的接口或类
    - 通用代码
    - 优点
      - 可以让两个没有任何关系的类在一起运行，只要适配器这个角色能够搞定他们就成
      - 增加了类的透明性
      - 提高了类的复用度
      - 灵活性非常好
    - 使用场景
      - 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合适的模式(即补救系统中的不足)
    - 注意事项
      - 它是为了解决正在服役的项目问题，而不是为了解决开发阶段的问题
    - 扩展
      - 对象适配器
        - 定义
          - 把target接口需要的所有的操作都委托给其他三个接口下的实现类，它的委托是通过对象层次的关联关系进行委托的，而不是继承关系
        - 通用代码
        - VS类适配器

- 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
- 理解
  - 主要用于接口的变换
- 桥梁模式
  - 定义
    - 将抽象和实现解耦，使得两者可以独立地变化(抽象角色引用实现角色，或者说抽象角色的部分实现是由实现角色完成的)
  - 通用代码
  - 优点
    - 抽象和实现分离
    - 优秀的扩充能力
    - 实现细节对高层代码透明
  - 使用场景
    - 不希望或不适用使用继承的场景
    - 接口或抽象类不稳定的场景
    - 重用性要求较高的场景
  - 注意事项
    - 当类的继承有N层的时候，可以考虑使用桥梁模式
  - 理解
    - 可替换的实现
- 组合模式
  - 定义
    - 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
  - 通用代码
  - 优缺点
    - 优点
      - 高层模块调用简单
      - 节点自由增加
    - 缺点
      - 限制了接口的影响范围，因为与依赖倒置原则冲突
  - 使用场景
    - 维护和展示部分-整体关系的场景，就是出现树形结构的场景
- 装饰模式
  - 定义
    - 动态地给一个对象添加一个写额外的职责。就增加功能来说，装饰模式相比生成子类更为灵活
  - 通用代码
  - 优缺点
    - 优点

- 装饰类和被装饰类可以独立发展，而不会相互耦合。Component类无须知道Decorator类，Decorator类是从外部来扩展Component类的功能，而Decorator也不用知道具体的构建
- 是继承关系的一个替代方案
- 装饰模式可以动态地扩展一个实现类的功能
- 缺点
  - 多层的装饰是比较复杂的
- 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，这些功能可以动态地撤销
  - 需要为一批的兄弟类进行改装或加装功能
- 理解
  - 增加额外的职责，不要和代理模式混淆

- 门面模式
  - 定义
    - 要求一个子系统的外部与内部的通信必须通过一个统一的对象进行。门面模式提供一个高层次的接口，使得子系统更易于使用
  - 通用代码
  - 优缺点
    - 优点(只能访问门面类)
      - 减少系统的相互依赖
      - 提高灵活性
        - 不用管子系统内部的变化
      - 提高安全性
    - 缺点
      - 不符合开闭原则
        - 要扩展时就得修改门面角色代码
  - 使用场景
    - 为一个复杂的模块或子系统提供一个供外界访问的接口
    - 子系统相对独立
      - 外界对子系统的访问只要黑箱操作即可
    - 预防低水平人员带来的风险扩散
  - 注意事项
    - 一个子系统可以有多个门面
      - 门面过于庞大
        - 拆分门面
      - 子系统可以提供不同的访问路径
        - 不同的模块访问的权限和内容不一样
    - 门面不参与子系统内的义务逻辑

- 享元模式

- 定义
  - 使用共享对象可有效地支持大量的细粒度的对象
- 对象状态
  - 内部状态(属性)
    - 内部状态是对象可共享出来的信息，存储在享元对象内部并且不会随环境改变而改变
  - 外部状态(key)
    - 对象得以依赖的一个标记，是随环境改变而改变的、不可以共享的状态
- 通用代码
- 优缺点
  - 优点
    - 大大减少应用程序创建的对象，降低程序内存的占用，增强程序的性能
  - 缺点
    - 提高了系统的复杂性，需要分离出外部状态和内部状态，而且外部状态具有固化特性，不随内部状态的改变而改变
- 使用场景
  - 系统中存在大量的相似对象
  - 细粒度的对象都具备较接近的外部状态，而且内部状态与环境无关
  - 需要缓冲池的场景
- 扩展
  - 线程安全问题
    - 对象池中的享元对象尽量多，躲到足够满足业务为止
  - 性能平衡
    - 使用自己编写的类作为外部状态，必须覆写equals方法和hashCode方法，而且执行效率还比较低，这种吃力不讨好的事情最好别做，外部状态最好以Java的基本类型作为标志，如String、int等，可以大幅地提升效率。

- 代理模式
  - 定义
    - 为其他对象提供一种代理以控制对这个对象的访问
  - 通用代码
  - 优点
    - 职责清晰
      - 真实的角色就是实现实际的业务逻辑，不用关心其他非本职责的事务，通过后期的代理完成一件事务，附带的结果就是编程简洁清晰
    - 高扩展性
      - 具体主题角色是随时都会发生变化的，只要它实现了接口，那我们的代理类完全就可以在不做任何修改的情况下使用
    - 智能化
  - 扩展
    - 普通代理
      - 定义

- 客户端只能访问代理角色，而不能访问真实角色
- 通用代码
- 强制代理
  - 定义
    - 要从真实的角色查找到代理角色，不允许直接访问真实角色或是代理
- 动态代理
  - 定义
    - 在实现阶段不用关心代理谁，而在运行阶段才指定代理哪个对象
  - 通用代码

- 行为型模式
  - 责任链模式
    - 定义
      - 使多个对象都有机会处理请求，从而避免了请求的发送者和接受者之间的耦合关系。将这些对象连成一条链，并沿着这条链传递该请求，直到有对象处理为止
    - 通用代码
    - 优缺点
      - 优点
        - 请求和处理分开
      - 缺点
        - 性能问题，每个请求都是从链头遍历到链尾，特别是在链比较常的时候，性能是个大问题
        - 调试不方便，特别是链条比较长，环节比较多的时候，由于采用了类似递归的方式，调试的时候逻辑可能比较复杂
    - 注意事项
      - 一定要控制链的长度，不能太长
  - 命令模式
    - 定义
      - 将一个请求封装成一个对象，从而让你使用不同的请求把客户端参数化，对请求排队或者记录请求日志，可以提供命令的撤销和恢复功能
    - 通用代码
    - 优缺点
      - 优点
        - 类间解耦
          - 调用者角色与接受者角色之间没有任何依赖关系，调用者实现功能时只需要调用command抽象类的excute方法就可以了
        - 可扩展性
          - Command的子类可以非常容易扩展，而调用者invoker和高层次的模块client不产生严重的代码耦合
        - 命令模式结合其他模式会更优秀

- 可以结合责任链模式，实现命令族解析任务；结合模板方法模式，减少command子类的膨胀问题
- 缺点
  - command的子类会膨胀非常大

- 解释器模式
  - 定义
    - 给定一门语言，定义它的文法的一种表示，并定义一个解释器，该解释器使用该表示来解释语言的句子
  - 通用代码
  - 优缺点
    - 优点
      - 扩展性好
    - 缺点
      - 解释器模式会引起类膨胀
      - 解释器模式采用地柜调用的方法
      - 效率低
  - 使用场景
    - 重复发生的问题
    - 简单语法需要解释的场景
  - 注意事项
    - 不要在重要的模块中使用
      - 因为维护困难
    - 使用频率低
      - 遇到“一种特定类型的问题发生的频率足够高”的情况，准备使用解释器模式时，可以考虑一下Expression4J、MESP（Math Expression String Parser）、Jep等开源的解析工具包（这三个开源产品都可以通过百度、Google搜索到，请读者自行查询），功能都异常强大，而且非常容易使用，效率也还不错，实现大多数的数学运算完全没有问题
- 迭代器模式
  - 定义
    - 提供一种方法访问一个容器对象中各个元素，而又不需要暴露该对象的内部细节
  - 使用java JDK中的Iterator，而不用自己写迭代器
- 中介者模式
  - 定义
    - 用一个中介对象封装一系列的对象交互，中介者使各个对象不需要显示地相互作用，从而使其耦合松散，而且可以独立地改变它们之间的交互
  - 通用代码
  - 优缺点
    - 优点
      - 减少类间的依赖，把原有的一对多的依赖变成了一对一的依赖，同事类只依赖中介者，减少了依赖，降低了类间的耦合

- 缺点
  - 中介者会膨胀得很大，而且逻辑复杂，原本N个对象直接相互依赖关系转换为中介者和同事类的依赖关系，同事类越多，中介者的逻辑就越复杂
- 使用场景
  - 中介者模式适用于多个对象之间相互紧密耦合的情况，紧密耦合的标准是：在类图中出现了蜘蛛网状结构。把蜘蛛网梳理为星型结构
- 实际应用
  - 机场调度中心
  - MVC框架
  - 媒体网关
  - 中介服务

- 备忘录模式
  - 定义
    - 在不破坏封装性的前提下，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
  - 通用代码
  - 使用场景
    - 需要保存和恢复数据的相关状态场景
    - 提供一个可回滚的操作
    - 需要监控的副本场景中
    - 数据库连接的食物管理就是用的备忘录模式
  - 注意事项
    - 备忘录的生命周期
      - 它创建出来的是要在最近的代码中使用，要主动管理它的生命周期，建立就要使用，不使用就要立刻删除其引用，等待垃圾回收器对它的回收处理
    - 备忘录的性能(不要在频繁建立备份的场景中使用)
      - 控制不了备忘录建立的对象的数量
      - 大对象的建立是要消耗资源的，系统的性能需要考虑
  - 扩展
    - clone方式的备忘录
      - 通用代码
      - 注意
        - 使用Clone方式的备忘录模式，可以使用在比较简单的场景或者比较单一的场景中，尽量不要与其他的对象产生严重的耦合关系
    - 多状态的备忘录模式
      - 通用代码
      - 注意
        - 如果要设计一个在运行期决定备份状态的框架，则建议采用AOP框架来实现，避免采用动态代理无谓地增加程序逻辑复杂性。
    - 多备份的备忘录

- 通用代码
- 注意
  - 内存溢出问题，该备份一旦产生就装入内存，没有任何销毁的意向，这是非常危险的。因此，在系统设计时，要严格限定备忘录的创建，建议增加Map的上限，否则系统很容易产生内存溢出情况
- 安全备忘录
  - 通用代码

- 观察者模式
  - 定义
    - 定义对象间一种一对多的依赖关系，使得每当一个对象改变状态，则所有依赖于它的对象都会得到通知并被自动更新
  - 通用代码
  - 优缺点
    - 优点
      - 观察者和被观察者之间是抽象耦合
      - 建立一套触发机制
    - 缺点
      - 开发效率和运行效率低下，因为这个是顺序执行的，所以一个观察者有问题，会阻塞其他观察者的运行
  - 使用场景
    - 关联行为场景
    - 时间多级触发场景
    - 跨系统的消息交换场景，例如消息队列的处理机制
  - 注意事项
    - 广播链的问题
      - 维护性差，建议消息最多转发一次
    - 异步处理问题
  - 扩展
    - 使用JDK中的Observable和Observer
    - 改造
      - 观察者和被观察者之间的消息沟通
      - 观察者响应方式
        - 多线程
        - 缓存技术
      - 被观察者尽量自己做主
    - 订阅发布模型
- 状态模式
  - 定义
    - 当一个对象内在状态改变时允许其改变行为，这个对象看起来像改变了其类
  - 通用代码

- 优缺点
  - 优点
    - 结构清晰
    - 遵循设计原则
    - 封装性好
  - 缺点
    - 子类会太多
- 使用场景
  - 行为随状态改变而改变的场景
  - 条件、分支判断语句的替代者
- 注意事项
  - 对象的状态最好不要超过5个

- 策略模式
  - 定义
    - 定义一组算法，将每个算法都封装起来，并且使他们之间可以互换
  - 通用代码
  - 优缺点
    - 优点
      - 算法可以自由切换
      - 避免使用多重条件判断
      - 扩展性良好
    - 缺点
      - 策略类数量增多
      - 所有的策略类都需要对外暴露(可以使用其他模式来修正这个缺陷，如工厂方法模式、代理模式或享元模式)
  - 使用场景
    - 多个类只有在算法或是行为上稍有不同的场景
    - 算法需要自由切换
    - 需要屏蔽算法规则
  - 注意事项
    - 系统中的一个策略家族的具体策略数量超过4个，则需要考虑使用混合模式
  - 扩展
    - 枚举策略

- 模板方法模式
  - 定义
    - 定义一个操作中的算法的架构，而将一些步骤延迟到子类中。使得子类可以不改变一个算法的结构即可重定义改算法的某些特定步骤。
  - 包含
    - 基本方法
      - 基本方法也叫基本操作，是由子类实现的方法，并且在模板方法中调用。

- 模板方法
  - 可以有一个或几个，一般是一个具体的方法，也就是一个框架，实现对基本方法的调度，完成固定的逻辑
- 通用代码
- 注意事项
  - 抽象模板中的基本方法尽量设计为protected类型，符合迪米特法则，不需要暴露的属性或方法尽量不要设置为protected类型。
- 优缺点
  - 优点
    - 封装不变部分，扩展可变部分
    - 提取公共部分代码，便于维护
    - 行为由父类控制，子类实现
  - 缺点
    - 由于子类对父类产生了影响，所以在复杂项目中会带来代码阅读的难度
- 应用场景
  - 多个子类共有的方法，并且逻辑基本相同时
  - 重要、复杂的算法，可以把核心算法设计为模板方法，周边的相关细节功能则由各个子类实现
  - 重构时，把相同的代码抽取到父类中，然后通过钩子函数约束其行为

- 访问者模式
  - 定义
    - 封装一些作用于某种数据结构的各元素的操作，它可以在不改变数据结构的前提下定义作用于这些元素的心的操作
  - 通用代码
  - 优缺点
    - 优点
      - 符合单一职责原则
      - 优秀的扩展性
      - 灵活性高
    - 缺点
      - 具体元素对访问者公布细节
      - 具体元素变更比较困难
      - 违背了依赖倒置原则
  - 使用场景
    - 一个对象结构包含很多对象，他们有不同的接口，想对这些对象实施一些依赖于其具体类的操作
    - 需要对一个对象中的对象进行很多不同并且不相关的操作，想避免让这些操作污染这些对象的类时
    - 业务规则要求遍历多个不同的对象
  - 扩展

- 统计功能
- 多个访问者
- 双分派