**Design Pattern**

我们常把 23 种经典的设计模式分为三类：创建型、结构型、行为型。实际上，设计模式要干的事情就是解耦。创建型模式是将创建和使用代码解耦，结构型模式是将不同功能代码解耦，行为型模式是将不同的行为代码解耦，具体到观察者模式，它是将观察者和被观察者代码解耦。

1. **创建型模式**: 主要解决对象的创建问题，封装复杂的创建过程，解耦对象的创建代码和使用代码
   1. 单例模式: Singleton

Use “static”key word, define a global object, avoid creating many object instance. It has lazy mode, and greedy mode

* 1. 工厂模式(Factory Design Pattern):工厂模式分为三种更加细分的类型：简单工厂、工厂方法和抽象工厂。不过，在 GoF 的《设计模式》一书中，它将简单工厂模式看作是工厂方法模式的一种特例，所以工厂模式只被分成了工厂方法和抽象工厂两类。工厂模式，它的作用无外乎下面这四个。这也是判断要不要使用工厂模式的最本质的参考标准。封装变化：创建逻辑有可能变化，封装成工厂类之后，创建逻辑的变更对调用者透明。代码复用：创建代码抽离到独立的工厂类之后可以复用。隔离复杂性：封装复杂的创建逻辑，调用者无需了解如何创建对象。控制复杂度：将创建代码抽离出来，让原本的函数或类职责更单一，代码更简洁。

When there has lots of related Classes, we can create a Factory class to contract all those object, and store them to a Map, then define a method to return different object base on the parameter.

* 1. 建造者模式 Builder： 工厂模式是用来创建不同但是相关类型的对象（继承同一父类或者接口的一组子类），由给定的参数来决定创建哪种类型的对象。建造者模式是用来创建一种类型的复杂对象，通过设置不同的可选参数，“定制化”地创建不同的对象。

When a Class has lots of the attributes need to define when it been create, and those attributes requirement are different for different caller. Then we can use Buider Design Pattern. When you new an object, you can “set” different attributes.

* 1. 原型模式 Prototype : 如果对象的创建成本比较大，而同一个类的不同对象之间差别不大（大部分字段都相同）可以利用对已有对象（原型）进行复制（或者叫拷贝）的方式来创建新对象，以达到节省创建时间的目的. 这种基于原型来创建对象的方式就叫作原型设计模式。实际上，创建对象包含的申请内存、给成员变量赋值这一过程，本身并不会花费太多时间，或者说对于大部分业务系统来说，这点时间完全是可以忽略的。应用一个复杂的模式，只得到一点点的性能提升，这就是所谓的过度设计，得不偿失。但是，如果对象中的数据需要经过复杂的计算才能得到（比如排序、计算哈希值），或者需要从 RPC、网络、数据库、文件系统等非常慢速的 IO 中读取，这种情况下，我们就可以利用原型模式，从其他已有对象中直接拷贝得到，而不用每次在创建新对象的时候，都重复执行这些耗时的操作。

When construct an object is complicated, e.g. need lots of calculation, sort. Tons of data load. We can use re-use the exited object as Prototype. Copy from other exiting object.

1. **结构型模式**：主要总结了一些类或对象组合在一起的经典结构
   1. 代理模式（Proxy Design Pattern）：在不改变原始类（或叫被代理类）代码的情况下，通过引入代理类来给原始类附加功能。 代理模式最常用的一个应用场景就是，在业务系统中开发一些非功能性需求，比如：监控、统计、鉴权、限流、事务、幂等、日志。我们将这些附加功能与业务功能解耦，放到代理类中统一处理，让程序员只需要关注业务方面的开发。代理模式在不改变原始类接口的条件下，为原始类定义一个代理类，主要目的是控制访问，而非加强功能，这是它跟装饰器模式最大的不同

Create a new class(proxy) to restrict or add new functionality. Similar implementation with Decorator Design Pattern

Spring AOP 是基于代理模式来实现的

* 1. 桥接模式（Bridge Design Pattern）：Decouple an abstraction from its implementation so that the two can vary independently。”->“将抽象和实现解耦，让它们可以独立变化。”桥接模式的目的是将接口部分和实现部分分离，从而让它们可以较为容易、也相对独立地加以改变。

Like JDBC driver, all kinds of Driver implement an interface java.sql.Driver. When class load. The driver e.g. MySQL driver get registered to DriverManager class. (bridge to the detail driver). then client can use DriverManager connect different DB.

* 1. 装饰器模式(Decorator Design Pattern): 第一个比较特殊的地方是：装饰器类和原始类继承同样的父类，这样我们可以对原始类“嵌套”多个装饰器类。第二个比较特殊的地方是：装饰器类是对功能的增强，这也是装饰器模式应用场景的一个重要特点。符合“组合关系”这种代码结构的设计模式有很多，比如之前讲过的代理模式、桥接模式，还有现在的装饰器模式。尽管它们的代码结构很相似，但是每种设计模式的意图是不同的。就拿比较相似的代理模式和装饰器模式来说吧，代理模式中，代理类附加的是跟原始类无关的功能，而在装饰器模式中，装饰器类附加的是跟原始类相关的增强功能。装饰者模式在不改变原始类接口的情况下，对原始类功能进行增强，并且支持多个装饰器的嵌套使用
  2. 适配器模式（Adapter Design Pattern）：适配器模式是一种事后的补救策略。适配器提供跟原始类不同的接口，而代理模式、装饰器模式提供的都是跟原始类相同的接口。
  3. 门面/外观模式(Facade Design Pattern): “Provide a unified interface to a set of interfaces in a subsystem. Facade Pattern defines a higher-level interface that makes the subsystem easier to use.””门面模式为子系统提供一组统一的接口，定义一组高层接口让子系统更易用。” 例子：利用门面模式，让后端服务器提供一个包裹 a、b、d 三个接口调用的接口 x。App 客户端调用一次接口 x，来获取到所有想要的数据，将网络通信的次数从 3 次减少到 1 次，也就提高了 App 的响应速度
  4. 组合模式(Composite Design Pattern): “Compose objects into tree structure to represent part-whole hierarchies.Composite lets client treat individual objects and compositions of objects uniformly.””将一组对象组织（Compose）成树形结构，以表示一种“部分 - 整体”的层次结构。组合让客户端（在很多设计模式书籍中，“客户端”代指代码的使用者。）可以统一单个对象和组合对象的处理逻辑”。这里讲的“组合模式”，主要是用来处理树形结构数据。这里的“数据”，你可以简单理解为一组对象集合。组合模式使用了树形的数据结构以及递归算法
  5. 享元模式(Flyweight Design Pattern): 所谓“享元”，顾名思义就是被共享的单元。享元模式的意图是复用对象，节省内存，前提是享元对象是不可变对象。代码结构。实际上，它的代码实现非常简单，主要是通过工厂模式，在工厂类中，通过一个 Map 来缓存已经创建过的享元对象，来达到复用的目的。

Integer a = new Integer(123);

Integer a = 123;

Integer a = Integer.valueOf(123);

第一种创建方式并不会使用到 IntegerCache，而后面两种创建方法可以利用 IntegerCache 缓存，返回共享的对象(享元)

1. **行为型**：主要解决的就是“类或对象之间的交互”问题
   1. 观察者模式（Observer Design Pattern）也被称为发布订阅模式（Publish-Subscribe Design Pattern）“Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.”“在对象之间定义一个一对多的依赖，当一个对象状态改变的时候，所有依赖的对象都会自动收到通知”
   2. 模板模式(Template Method Design Pattern): “Define the skeleton of an algorithm in an operation, deferring some steps to subclasses. Template Method lets subclasses redefine certain steps of an algorithm without changing the algorithm`s structure.””模板方法模式在一个方法中定义一个算法骨架，并将某些步骤推迟到子类中实现。模板方法模式可以让子类在不改变算法整体结构的情况下，重新定义算法中的某些步骤。” 模板模式把一个算法中不变的流程抽象到父类的模板方法 templateMethod() 中，将可变的部分 method1()、method2() 留给子类 ContreteClass1 和 ContreteClass2 来实现。所有的子类都可以复用父类中模板方法定义的流程代码。 普通的函数调用，回调是一种双向调用关系。A 类事先注册某个函数 F 到 B 类，A 类在调用 B 类的 P 函数的时候，B 类反过来调用 A 类注册给它的 F 函数。这里的 F 函数就是“回调函数”。A 调用 B，B 反过来又调用 A，这种调用机制就叫作“回调”
   3. 策略模式(Strategy Design Pattern): “Define a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithm vary independently from clients that use it.””定义一族算法类，将每个算法分别封装起来，让它们可以互相替换。策略模式可以使算法的变化独立于使用它们的客户端（这里的客户端代指使用算法的代码）”策略模式用来解耦策略的定义、创建、使用。实际上，一个完整的策略模式就是由这三个部分组成的。

* 策略类的定义比较简单，包含一个策略接口和一组实现这个接口的策略类。
* 策略的创建由工厂类来完成(Map来缓存策略)，封装策略创建的细节。
* 策略模式包含一组策略可选，客户端代码如何选择使用哪个策略，有两种确定方法：编译时静态确定和运行时动态确定。其中，“运行时动态确定”才是策略模式最典型的应用场景。
  1. 职责链模式(Chain Of Responsibility Design Pattern): “Avoid coupling the sender of a request to its receiver by giving more than one object a chance to handle the request. Chain the receiving objects and pass the request along the chain until an object handles it.””将请求的发送和接收解耦，让多个接收对象都有机会处理这个请求。将这些接收对象串成一条链，并沿着这条链传递这个请求，直到链上的某个接收对象能够处理它为止。”

Like UGC(User Generated Content), we need to filter user content. We can define a Chain of Responsibility. Each of filter is a class, add the filter object to the chain(can be LinkedList or ArrayList). in the framework has code look through the chain. It will be very easy add/modify filters. Sample: Tomcat Servlet Filter, Spring Interceptor

ObjectUtils.isEmpty

* 1. 状态模式(State Design Pattern): 状态模式是状态机的一种实现方式。针对状态机，今天我们总结了三种实现方式。第一种实现方式叫分支逻辑法。利用 if-else 或者 switch-case 分支逻辑，参照状态转移图，将每一个状态转移原模原样地直译成代码。对于简单的状态机来说，这种实现方式最简单、最直接，是首选。第二种实现方式叫查表法。对于状态很多、状态转移比较复杂的状态机来说，查表法比较合适。通过二维数组来表示状态转移图，能极大地提高代码的可读性和可维护性。第三种实现方式叫状态模式。对于状态并不多、状态转移也比较简单，但事件触发执行的动作包含的业务逻辑可能比较复杂的状态机来说，我们首选这种实现方式。
  2. 迭代器模式（Iterator Design Pattern)也叫作游标模式（Cursor Design Pattern）它用来遍历集合对象。这里说的“集合对象”也可以叫“容器”。遍历集合一般有三种方式：for 循环、foreach 循环、迭代器遍历。后两种本质上属于一种，都可以看作迭代器遍历。相对于 for 循环遍历，利用迭代器来遍历有下面三个优势：迭代器模式封装集合内部的复杂数据结构，开发者不需要了解如何遍历，直接使用容器提供的迭代器即可；迭代器模式将集合对象的遍历操作从集合类中拆分出来，放到迭代器类中，让两者的职责更加单一；迭代器模式让添加新的遍历算法更加容易，更符合开闭原则。除此之外，因为迭代器都实现自相同的接口，在开发中，基于接口而非实现编程，替换迭代器也变得更加容易
  3. 访问者模式(Visitor Design Pattern): “Allows for one or more operation to be applied to a set of objects at runtime, decoupling the operations from the object structure.””允许一个或者多个操作应用到一组对象上，解耦操作和对象本身。”
  4. 备忘录模式（Memento Design Pattern）也叫快照（Snapshot）模式。“Captures and externalizes an object’s internal state so that it can be restored later, all without violating encapsulation.”“在不违背封装原则的前提下，捕获一个对象的内部状态，并在该对象之外保存这个状态，以便之后恢复对象为先前的状态。”

Sample: use a Stack to keep/snapshot of an object before making any changes. Restore/undo will get value from Stack.

* 1. 命令模式、
  2. 解释器模式、
  3. 中介模式