设计模式是对代码的总结和规范化。

某种程度上是人与人之间进行的关于代码的信息交换标准。比如你跟我说“代理模式”，只传输了四个字，但在我的脑海里已出现静态代理、动态代理，动态代理可能是jdk类型的，也可能是增强字节码的。不管是何种类型，这些代理对象的出现都只有一个目的，代替原对象进行工作。至于为什么会出现代理模式，因为我们希望自己的代码浅显易懂，只关注那部分关键的逻辑，至于与逻辑无关的部分，我们希望他们能以一种组合的方式把他们并在一起去执行。有了这样一个“标准”，交流起来也方便多了，当然也能够更快更彻底的理解别人写的代码了。

前面我说的设计模式是什么样什么样的信息交换标准，其实也就是规范化的意思。那为什么说它是对代码的总结呢？总结实际上就是把共同特性抽出来，也就是一个精炼的过程。比如我们需要一个对象，可能会直接new一个出来，也可能希望它是一个局部变量，或者仅供线程使用的，抑或是给整个app使用的，甚至是给整台机器使用以及给外部机器使用的。

不管怎样，这些都属于创建对象的范畴，共同特征是创建，属于创建型模式。针对具体的需求，我们又有更细分的设计模式，比如app范围类仅有一个实例的单例模式，根据一个实例产生另一个拥有相同数据的原型模式。

设计模式有23种，分为 创建型，结构型，行为型。

创建型是指一个对象是如何被创建出来的。假设有人问小孩是怎么出来的，相信他期待的答案不是生出来的。小孩可能是顺产的，可能是剖腹产的，还可能是早产的。他肯定希望得到包括生出来时的细节。现在回到对象的创建上，我们关注的是创建的细节，这些细节包含了产生对象具有什么特征，产生的对象有什么用途，产生的过程。

工厂方法和抽象工厂模式产生的对象都是实际运行时确定的，比如工厂要生产武器，与现实不同，你只有在拿到工厂递给你的武器时才知道它是一把刀还是剑还是什么，但这些可能性具有共同的特征，就是它们都是武器，都能攻击。

建造者模式是把创建对象的麻烦过程给抽出来，假设一对夫妻要一个小孩，这个丈夫只需要提供什么你懂的，妻子呢，可以看成一个建造者模式，经历漫长的孕期和痛苦的分娩，最后孩子平安地落到了丈夫的手上。这里丈夫就是客户端，提供一些“参数”，妻子就是建造者模式，把创建对象的复杂过程全程包办了。

原型模式强调的是需要一个和现有的某个对象具有相同数据的对象。就建造者模式来说，父亲再提供一次“参数”，生出来的小孩可能长相高度等等会不一样。但如果我需要和之前出生的小孩一模一样的孩子怎么办？只能克隆了，就算是克隆，因为环境的不一样，也许他们的聪明程度都不一样，因为人很复杂，属性很多，还能通过学习改变自身属性。而在简单的计算机应用里，是能够完全克隆出一个具有相同数据的对象来的。

单例模式是指我们需要的那种对象，数量是受限的。比如办公室里有一台打印机，把打印机看成对象，把办公室成员看成客户端，因为打印机太耗钱，不见得人手一台，所以只能所有成员共用一台来节省开支。由于打印机是消耗品，你对它的使用会对它造成损耗，损耗实际上是内部状态发生了改变，如果多个办公室成员同时对打印机进行“损耗”，就会产生并发问题，甚至会因此吵起来，根源就是实际资源数量过少引起的。如果办公室来了一位美女，我们把该美女看成单例，这个单例对象所提供的方法只有——供人欣赏。那么我们欣赏她是不会“损耗”她的，办公室所有人同时观看她都不会有问题，因为这个“单例对象”提供的行为是只读的，没有涉及到修改。当然单例的产生过程也会有并发问题，比如你想使用打印机时，主任说现在还没有，让你赶紧去买一台，恰好主任手机没挂，被电话里的小明听到了，结果就是办公室有了两台打印机，其原因就是产生“单例”的两个“线程”没有同步信息的原因，假设他们的购买时间差差距很大也不至于产生这样的结果。

如果把类比作菜谱，那么创建型模式就是根据菜谱做出菜的过程。例如建造者模式就是——参数“我要吃这个”，你指了指某道菜，“建造者”厨师就马上为你做好端了上来。

那么结构型模式相当于如何把菜谱结合在一起，使做出的菜更符合刁钻客人的胃口。

比如你想吃菜谱上的“澳洲大龙虾”，实际我们店里没有这道菜，它是我们的“代理菜”，

它的“真实菜谱”可能是隔壁海鲜店的“盐焗超大龙虾”，在制作“澳洲大龙虾”菜的过程中，得和隔壁海鲜店交流，等他们做好拿到我们这边来。到最后端到你面前的时候，你都觉得这就是你所需要的，我们店制作的“澳洲大龙虾”。其实，我们不过是代理罢了。

结构型中把类比作菜谱，把对象比作做出来的菜，在行为型中作这种比喻不再合适，因为在行为型中，“做出来的菜”不再是死的任人宰割的，而是活的，它和其它被“做出来的菜”有一定的关系，没有贴近现实。行为型强调的是对象之间的关系，类似于任何人的关系，很自然地贴近现实。

责任链，拿浏览器例子来说，假设输入https://www.baidu.com/s?wd=设计模式，对于我来说，我期待的是显示“设计模式”的搜索结果。那么究竟经历发生了怎样一个过程？它可能经历了一个转码的过程，完成编码集的转换；可能经历了一个登陆认证的过程，若我已登陆，给我显示我定制的背景；还可能经历一个浏览器验证，发现如果我不是浏览器请求，比如是用代码推送的爬虫请求，就拒绝为我服务。如果这一系列的处理如果放到一个方法里来做，显然太过庞大。那么把他们分散出去，放到不同的方法中，再以链的方式，一个一个地执行这些方法，就等于执行了所有的逻辑。每个链中的方法都可以选择继续传到下一个链来处理，或是直接驳回你的请求，返回404给你。从用户的角度来看，并不知道这个链的存在，就算知道，也无法断定链中各元素顺序是怎样的。开发者现在只要在链中某处加入一个新的方法，就为整个请求处理过程加入了新的特性。

迭代器模式，一个医生来到教室外要给学生打预防针，教室里有很多学生，他们座位排列如何医生是不得而知的，他只知道自己叫“下一个”时，就会从教室门口走出一个心惊胆战的学生出来，走到自己面前把袖子撩起来。对于医生来说，你教室里两列两列这样的排，或者排成一个圈都无所谓，只要我叫“下一个”时你们一次只出来一个，别捣乱就行了。当然，打过的也别再来了，害怕的也必须出来，这些由你们的班主任负责调控。这个教室对外提供的这种顺序访问内部学生的行为就是迭代器模式。它并不会暴露内部座位如何排列的，教室内是不是还有家长、其他老师等等，班主任只会督促让那些需要打针的学生走出去让医生为其打针。

解释器模式：

策略模式指定义了不同的算法，并且可以互相替换使用。例如有个没感情没主见的小孩，视为——上下文环境，每天的行为视为——策略，

你让他周一至周五好好学习，周末好好玩，这样他就根据每天的时间采取相应的策略。然而几天后他就进了医院，因为他24小时都在“好好学习”，没撑过几天就倒下了。

状态模式指上下文内维护了一个状态，状态改变时允许改变其行为，跟策略模式又相似处。现在又有个有感情有主见的人类小孩，视为上下文环境，他有自己的心情、疲惫度属性，这些都有各自的字段存储，视为状态。然后你也让他周一至周五好好学习，周末好好玩。等到了周一，上午八点，他的状态——“干劲十足”，心情=开心，疲惫=精神焕发，此时他的行为表现为高效率学习ing。

随着不断地学习，他的状态在十点切换成——“还有多久吃午饭”，心情=有点开心，疲惫=一般，此时他的行为表现为学习中，但有点走神。

最后状态变成——“累，坐等午饭”，心情=一般，疲惫=很累，此时他的行为表现为趴桌上睡觉。仅仅是周一上午，就发生了这么多的状态切换，还可以引入饥饿度属性，当该值增高到一定程度切换到“吃饭”模式，吃完又到“干劲十足”状态。不同状态对应的行为可以视为策略模式中的策略，不过状态模式下的策略，是可以根据上下文环境自动切换的，它们的实现类中有切换策略这个方法。而策略模式中，就算你让他“周一至周五适当地学习，周末适当地玩”，你所说的适当，是指身体累了就不要“学习”/"玩"了，但他（上下文中）没有心情、疲惫度等状态属性，无法理解“适当”一词的意思，给你来个随机切换“玩”“学习”模式。最后他还是倒下了，因为在我的例子中，他没有人类的感情状态属性，但身体是人类的，结果他只顾着适当地切换“玩”“学习”模式，没有休息，把他作为人类的身体累跨了。

访问者模式，不改变各元素的前提下，定义一个作用于对象中各元素的新操作。一个教室可以视为一个目标对象，教室中的成员视为目标对象的各个元素，这个教室可以接受一个访问者，教室接收访问者时的具体行为是带这个访问者在教室走一圈看一眼各个学生，然后领他出去。如果现在要统计教室内人数，室内男生人数，女生人数，一般是在教室这个类中实现的。但为了保证OCP原则，不能修改教室类和内部元素类，那么只能通过访问者实现这些方法了。可以在访问者内部定义人数、男生数、女生数三个变量，在教室走一圈这个过程中，也就是观察每个学生的过程，人数++，然后根据是男是女对对应的变量++；最后走完一圈出去后，通过相应方法打印这些统计结果便可。现在若又要统计长得漂亮的女生，只要判断现在观察的对象是否满足性别=女&&长得漂亮=是，若满足则对应统计字段++，访问完输出即可。也就是说，对于不同的访问者，他们到教室里的行为都是一样，由教室这个类决定，就是在里面走一圈观察每一个学生。至于他们出教室后获得什么信息，全在于他们的关注点在哪。上面的例子还能分开，分为3个不同的总人数访问者，男生总数访问者，女生总数访问者，然后由教室对象分别接受这3个访问者即可。