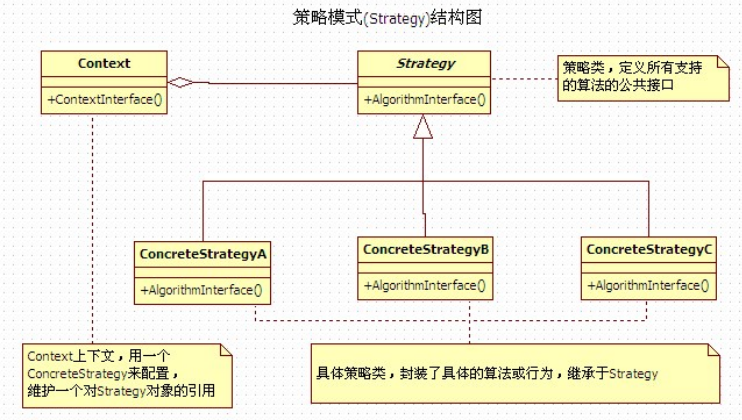
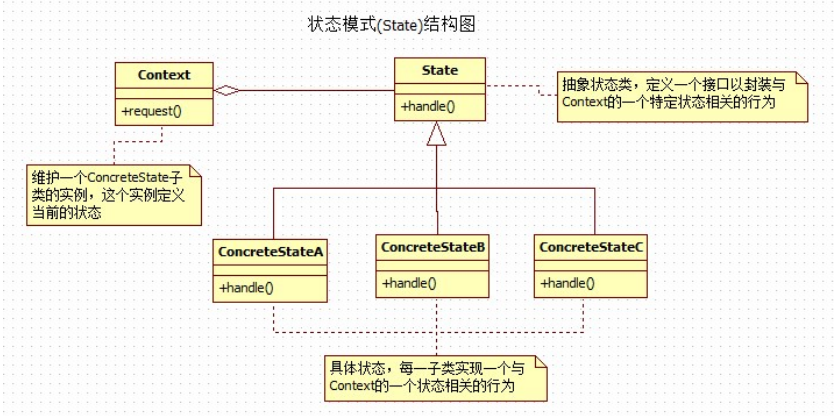
【设计模式】策略模式与状态模式。

策略模式与状态模式在实现上有共同之处，都是把不同的情形抽象为统一的接口来实现，就放在一起进行记录。2个模式的UML建模图基本相似，区别在于状态模式需要在子类实现与context相关的一个状态行为。

状态模式的的思想是，状态之间的切换，在状态A执行完毕后自己控制状态指向状态B。状态模式是不停的切换状态执行。

策略模式的思想上是，考虑多种不同的业务规则将不同的算法封装起来，便于调用者选择调用。策略模式只是条件选择执行一次。





策略模式

1. Strategy：定义所有支持的算法的公共接口抽象类.
2. ConcreteStrategy：封装了具体的算法或行为，继承于Strategy
3. Context：用一个ConcreteStrategy来配置，维护一个对Strategy对象的引用。

状态模式

1. State：抽象状态类，定义一个接口以封装与context的一个状态相关的行为
2. ConcreteState：具体状态，每一子类实现一个与Context的一个状态相关的行为
3. Context：维护一个ConcreteState子类的实例，这个实例定义当前的状态。

使用场景：

状态模式主要解决的是当控制一个对象状态转换的条件表达式过于复杂时的情况。把状态的判断逻辑转移到表示不同状态的一系列类当中，可以把复杂的判断逻辑简化。当一个对象的行为取决于它的状态，并且它必须在运行时刻根据状态改变它的行为时，就可以考虑使用状态模式了。

策略模式的Strategy类层次为Context定义了一系列的可供重用的算法或行为。继承有助于析取出这些算法中的公共功能。在实践中，我们发现可以用它来封装几乎任何类型的规则，只要在分析过程中听到需要在不同时间应用不同的业务规则，就可以考虑使用策略模式处理这种变化的可能性。

状态模式和策略模式的比较

两个模式的实现类图虽然一致，但是实现目的不一样！

首先知道，策略模式是一个接口的应用案例，一个很重要的设计模式，简单易用，策略模式一般用于单个算法的替换，客户端事先必须知道所有的可替换策略，由客户端去指定环境类需要哪个策略，注意通常都只有一个最恰当的策略(算法)被选择。其他策略是同级的，可互相动态的在运行中替换原有策略。

而状态模式的每个状态子类中需要包含环境类(Context)中的所有方法的具体实现——条件语句。通过把行为和行为对应的逻辑包装到状态类里，在环境类里消除大量的逻辑判断，而不同状态的切换由继承(实现)State的状态子类去实现，当发现修改的当前对象的状态不是自己这个状态所对应的参数，则各个状态子类自己给Context类切换状态(有职责链模式思想)！且客户端不直接和状态类交互，客户端不需要了解状态！(和策略不一样)，策略模式是直接依赖注入到Context类的参数进行选择策略，不存在切换状态的操作，客户端需要了解策略！

联系：状态模式和策略模式都是为具有多种可能情形设计的模式，把不同的处理情形抽象为一个相同的接口(抽象类)，符合对开闭原则，且策略模式更具有一般性，在实践中，可以用策略模式来封装几乎任何类型的规则，只要在分析过程中听到需要在不同实践应用不同的业务

规则，就可以考虑使用策略模式处理，在这点上策略模式是包含状态模式的功能的。

小结：状态模式的使用场景是什么？

状态模式主要解决的是(目的or意图)：控制一个对象内部的状态转换的条件表达式过于复杂时的情况，且客户端调用之前不需要了解具体状态。它把状态的判断逻辑转到表现不同状态的一系列类当中，可以把复杂的判断逻辑简化。维持开闭原则，方便维护

，还有重要一点下面会总结，**状态模式是让各个状态对象自己知道其下一个处理的对象是谁！即在状态子类编译时在代码上就设定好了**！

状态模式的优缺点都是什么？

优点，前面说了很多了.....

- 状态模式使得代码中复杂而庸长的逻辑判断语句问题得到了解决，而且状态角色将具体的状态和他对应的行为及其逻辑判断封装了起来，这使得增加一种新的状态显得十分简单。
- 把容易出错的if-else语句在环境类 or 客户端中消除，方便维护。
- 每一个状态类都符合"开闭"原则——对状态的修改关闭，对客户端的扩展开放，可以随时增加新的Person的状态，或者删除。
- State类在只有行为需要抽象时，就用接口，有其他共同功能可以用抽象类，这点和其他一些(策略)模式类似。

缺点：

使用状态模式时，每个状态对应一个具体的状态类，使结构分散，类的数量变得很多！使得程序结构变得稍显复杂，阅读代码时相对之前比较困难，不过对于优秀的研发人员来说，应该是微不足道的。因为想要获取弹性！就必须付出代价！除非我们的程序是一次性的！用完就丢掉.....如果不是，那么假设有一个系统，某个功能需要很多状态，如果不使用状态模式优化，那么在环境类(客户端类)里会有大量的整块整块的条件判断语句！

Strategy模式有下面的一些优点：
1) 相关算法系列 Strategy类层次为Context定义了一系列的可供重用的算法或行为。继承有助于析取出这些算法中的公共功能。
2) 提供了可以替换继承关系的办法：继承提供了另一种支持多种算法或行为的方法。你可以直接生成一个Context类的子类，从而给它以不同的行为。但这会将行为硬行编制到 Context中，而将算法的实现与Context的实现混合起来,从而使Context难以理解、难以维护和难以扩展，而且还不能动态地改变算法。最后你得到一堆相关的类，它们之间的唯一差别是它们所使用的算法或行为。将算法封装在独立的Strategy类中使得你可以独立于其Context改变它，使它易于切换、易于理解、易于扩展。
3) 消除了一些if else条件语句 ：Strategy模式提供了用条件语句选择所需的行为以外的另一种选择。当不同的行为堆砌在一个类中时，很难避免使用条件语句来选择合适的行为。将行为封装在一个个独立的Strategy类中消除了这些条件语句。含有许多条件语句的代码通常意味着需要使用Strategy模式。
4) 实现的选择 Strategy模式可以提供相同行为的不同实现。客户可以根据不同时间 /空间权衡取舍要求从不同策略中进行选择。

Strategy模式缺点：

1)客户端必须知道所有的策略类，并自行决定使用哪一个策略类：本模式有一个潜在的缺点，就是一个客户要选择一个合适的Strategy就必须知道这些Strategy到底有何不同。此时可能不得不向客户暴露具体的实现问题。因此仅当这些不同行为变体与客户相关的行为时，才需要使用Strategy模式。
2) Strategy和Context之间的通信开销 ：无论各个ConcreteStrategy实现的算法是简单还是复杂，它们都共享Strategy定义的接口。因此很可能某些 ConcreteStrategy不会都用到所有通过这个接口传递给它们的信息；简单的 ConcreteStrategy可能不使用其中的任何信息！这就意味着有时Context会创建和初始化一些永远不会用到的参数。如果存在这样问题，那么将需要在Strategy和Context之间更进行紧密的耦合。
3)策略模式将造成产生很多策略类：可以通过使用享元模式在一定程度上减少对象的数量。增加了对象的数目 Strategy增加了一个应用中的对象的数目。有时你可以将 Strategy实现为可供各Context共享的无状态的对象来减少这一开销。任何其余的状态都由 Context维护。Context在每一次对Strategy对象的请求中都将这个状态传递过去。共享的 Strategy不应在各次调用之间维护状态。