在本章中，我们将要学习Strategy模式。

【整体地替换算法】

Strategy的意思是“策略”，指的是与敌军对垒时行军作战的方法。在编程中，我们可以将它理解为“算法”。

无论什么程序，其目的都是解决问题。而为了解决问题，我们又需要编写特定的算法。使用Strategy模式可以整体地替换算法的实现部分。能够整体地替换算法，能让我们轻松地以不同的算法去解决同一个问题，这种模式就是Strategy模式。

|10.2示例程序

下面我们来看一段使用了Strategy模式的示例程序。这段示例程序的功能是让电脑玩“猜拳”

游戏。

我们考虑了两种猜拳的策略。第一种策略是“如果这局猜拳获胜，那么下一局也出一样的手势”（Winningstrategy），这是一种稍微有些笨的策略；另外一种策略是“根据上一局的手势从概率上计算出下一局的手势”（ProbStrategy）。

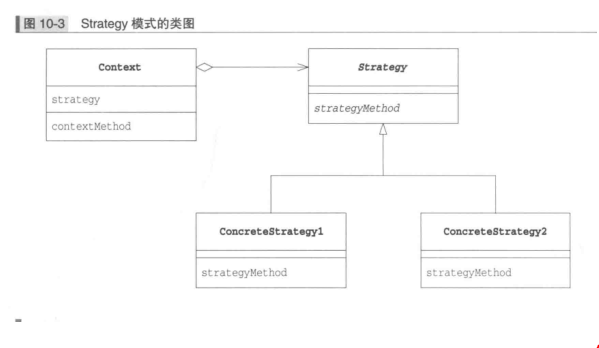
|10.3 Strategy模式中的登场角色

在Strategy模式中有以下登场角色。

◆Strategy（策略）Strategy角色负责决定实现策略所必需的接口（API）。在示例程序中，由strategy接口扮演此角色。

·ConcreteStrategy（具体的策略）ConcreteStrategy角色负责实现Strategy角色的接口（API），即负责实现具体的策略（战略、方向、方法和算法）。在示例程序中，由winningstrategy 类和ProbStrategy类扮演此角色。

·Context（上下文）负责使用Strategy角色。Context角色保存了ConcreteStrategy角色的实例，并使用ConcreteStrategy角色去实现需求（总之，还是要调用Strategy角色的接口（APl））。在示例程序中，由Player类扮演此角色。



|10.4拓展思路的要点

|为什么需要特意编写Strategy角色

通常在编程时算法会被写在具体方法中。Strategy模式却特意将算法与其他部分分离开来，只是定义了与算法相关的接口（APl），然后在程序中以委托的方式来使用算法。

这样看起来程序好像变复杂了，其实不然。例如，当我们想要通过改善算法来提高算法的处理速度时，如果使用了Strategy模式，就不必修改Strategy角色的接口（API）了，仅仅修改ConcreteStrategy角色即可。而且，使用委托这种弱关联关系可以很方便地整体替换算法。例如。

如果想比较原来的算法与改进后的算法的处理速度有多大区别，简单地替换下算法即可进行测试。

使用Strategy模式编写象棋程序时，可以方便地根据棋手的选择切换Al例程的水平。

|程序运行中也可以切换策略

如果使用Strategy模式，在程序运行中也可以切换ConcreteStrategy角色。例如，在内存容量少的运行环境中可以使用SlowButLessMemoryStrategy（速度慢但省内存的策略），而在内存容量多的运行环境中则可以使用FastButMoreMemoryStrategy（速度快但耗内存的策略）。

此外，还可以用某种算法去“验算”另外一种算法。例如，假设要在某个表格计算软件的开发版本中进行复杂的计算。这时，我们可以准备两种算法，即“高速但计算上可能有Bug的算法”和

“低速但计算准确的算法”，然后让后者去验算前者的计算结果。

|10.5相关的设计模式

◆Flyweight模式（第20章）有时会使用Flyweight模式让多个地方可以共用ConcreteStrategy角色。

◆Abstract Factory模式（第8章）使用Strategy模式可以整体地替换算法。

使用Abstract Factory模式则可以整体地替换具体工厂、零件和产品。

◆State模式（第19章）使用Strategy模式和State模式都可以替换被委托对象，而且它们的类之间的关系也很相似。

但是两种模式的目的不同。

在Strategy模式中，ConcreteStrategy角色是表示算法的类。在Strategy模式中，可以替换被委托对象的类。当然如果没有必要，也可以不替换。

而在State模式中，ConcreteState角色是表示“状态”的类。在State模式中，每次状态变化时，被委托对象的类都必定会被替换。