## 9. 策略（Strategy）

### Intent

定义一系列算法，封装每个算法，并使它们可以互换。

策略模式可以让算法独立于使用它的客户端。

### Class Diagram

* Strategy 接口定义了一个算法族，它们都实现了 behavior() 方法。
* Context 是使用到该算法族的类，其中的 doSomething() 方法会调用 behavior()，setStrategy(Strategy) 方法可以动态地改变 strategy 对象，也就是说能动态地改变 Context 所使用的算法。

### 与状态模式的比较

状态模式的类图和策略模式类似，并且都是能够动态改变对象的行为。但是状态模式是通过状态转移来改变 Context 所组合的 State 对象，而策略模式是通过 Context 本身的决策来改变组合的 Strategy 对象。所谓的状态转移，是指 Context 在运行过程中由于一些条件发生改变而使得 State 对象发生改变，注意必须要是在运行过程中。

状态模式主要是用来解决状态转移的问题，当状态发生转移了，那么 Context 对象就会改变它的行为；而策略模式主要是用来封装一组可以互相替代的算法族，并且可以根据需要动态地去替换 Context 使用的算法。

### Implementation

设计一个鸭子，它可以动态地改变叫声。这里的算法族是鸭子的叫声行为。

public interface QuackBehavior {
 void quack();
}

public class Quack implements QuackBehavior {
 @Override
 public void quack() {
 System.out.println("quack!");
 }
}

public class Squeak implements QuackBehavior{
 @Override
 public void quack() {
 System.out.println("squeak!");
 }
}

public class Duck {

 private QuackBehavior quackBehavior;

 public void performQuack() {
 if (quackBehavior != null) {
 quackBehavior.quack();
 }
 }

 public void setQuackBehavior(QuackBehavior quackBehavior) {
 this.quackBehavior = quackBehavior;
 }
}

public class Client {

 public static void main(String[] args) {
 Duck duck = new Duck();
 duck.setQuackBehavior(new Squeak());
 duck.performQuack();
 duck.setQuackBehavior(new Quack());
 duck.performQuack();
 }
}

squeak!
quack!

### JDK

* java.util.Comparator#compare()
* javax.servlet.http.HttpServlet
* javax.servlet.Filter#doFilter()