Factory

当client不知道要创建哪个具体类的实例，或者不想在client代码中指明要具体 创建的实例时，用工厂方法。

客户端代码

```
public class Client {
    public static void main(String[] args) {
        Phone huawei = new HuaweiFactory().createPhone();
        huawei.call();
        huawei.message();
        Phone iphone = new IphoneFactory().createPhone();
        iphone.call();
        iphone.message();
    }
}
```

手机接口，用于组织手机

```
public interface Phone {
    public void call();
    public void message();
}
```

```
public class Huawei implements Phone{
    private final String name = "Huawei";

    public Huawei() {
        System.out.println("Huawei is created");
    }

    @Override
    public void call() {
        System.out.println("Calling from " + name);
    }
    @Override
    public void message() {
        System.out.println("Messaging from " + name);
    }
}
```

```
public class Iphone implements Phone{
    private final String name = "Iphone";

    public Iphone() {
        System.out.println("Iphone is created");
    }

    @Override
    public void call() {
        System.out.println("Calling from " + name);
    }
```

```
    @Override
    public void message() {
        System.out.println("Messaging from " + name);
    }
}
```

工厂接口，用于组织工厂

```
public interface Factory {
    public Phone createPhone();
}
```

```
public class IphoneFactory implements Factory{
    @Override
    public Phone createPhone() {
        return new Iphone();
    }
}
```

```
public class HuaweiFactory implements Factory{
    @Override
    public Phone createPhone() {
        return new Huawei();
    }
}
```

如果不用工厂设计模式，客户端代码为

```
public class Client {
    public static void main(String[] args) {
        Phone huawei = new Huawei();
        huawei.call();
        huawei.message();
        Phone iphone = new Iphone();
        iphone.call();
        iphone.message();
    }
}
```

这样的话客户端需要知道所需类的构造函数，这样的话客户端与实际的源代码关联就比较紧密，耦合性比较大

比如说如果又多了pad接口，并且有两个实现类：Ipad，HuaweiPad。利用工厂设计模式时只需要在Factory接口中加上createPad方法，并且再在IphoneFactory和HuaweiFactory中override就行了，客户端行为不变。如果不用的话客户端还要专门调用pad的构造方法，比较麻烦。

Adaptor

Adaptor设计模式是一种结构型设计模式，用于将一个类的接口转换成客户端所期望的另一个接口。这种模式可以让原本由于接口不兼容而无法一起工作的类能够协同工作。

原方法

```java
public class LegacyRectangle {
    public void display(int x1, int y1, int w, int h) {
        System.out.println("标准矩形");
    }
}
```

用户期望接口

```java
public interface Shape {
    void display(int x1, int y1, int x2, int y2);
}
```

“转接头”

```java
public class Rectangle implements Shape{
    @Override
    public void display(int x1, int y1, int x2, int y2) {
        new LegacyRectangle().display(x1, y1, x2 - x1, y2 - y1);
    }
}
```

就是起到一个转接头的作用，主要是处理用户端参数输入和已有方法参数列表不同的方法

Decorator

```java
// 定义组件接口
public interface Shape {
    void draw();
}

// 定义组件实现类
public class Rectangle implements Shape {
    @Override
    public void draw() {
        System.out.println("画一个矩形");
    }
}

// 定义装饰器抽象类
public abstract class ShapeDecorator implements Shape {
    protected Shape decoratedShape;

    public ShapeDecorator(Shape decoratedShape) {
        this.decoratedShape = decoratedShape;
    }

    public void draw() {
        decoratedShape.draw();
    }
}

// 定义具体装饰器类
public class RedShapeDecorator extends ShapeDecorator {
```

```
    public RedShapeDecorator(Shape decoratedShape) {
        super(decoratedShape);
    }

    @Override
    public void draw() {
        decoratedShape.draw();
        setRedBorder(decoratedShape);
    }

    private void setRedBorder(Shape decoratedShape) {
        System.out.println("添加红色边框");
    }
}
```

注意：

1. 基本装饰类是一个抽象类
2. 其中的属性是protected（为了它的子类能够直接调用该属性的方法）

```
public class Client {
    public static void main(String[] args) {
        // 使用普通的
        Shape s1 = new Rectangle();
        // 使用装饰的
        Shape s2 = new RedShapeDecorator(new Rectangle);
    }
}
```

Strategy

感觉就是用接口管理函数的调用，然后再用委托来调用函数

```
// 策略接口
public interface SortStrategy {
    void sort();
}

// 具体策略
public class QiuckSort implements SortStrategy{
    @Override
    public void sort() {
        System.out.println("快速排序");
    }
}

public class MergeSort implements SortStrategy{
    @Override
    public void sort() {
        System.out.println("归并排序");
    }
}

// 使用算法的类
```

```
public class Student {
    public void sort(SortStrategy s) {
        s.sort;
    }
}
```

Template Method

模板方法设计模式的核心是一个抽象类，**它定义了一个算法的骨架，将一些步骤延迟到子类中实现。**这个抽象类包含一个或多个抽象方法，这些方法由子类实现，以便在不改变算法结构的情况下重新定义算法的某些步骤。这个抽象类还可以包含具体方法，这些方法在算法中的多个步骤中都使用到了，但是可以被子类重写。

```
// 抽象类
abstract class Character {
    // 算法骨架
    public void attack() {
        System.out.println("准备攻击");
        System.out.println("使用 " + getWeapon() + " 攻击");
        System.out.println("攻击完成");
    }
    // protected作用范围：同一个包，不同包的子孙类可不可以用要说明
    // 算法的一部分
    protected abstract String getWeapon();
}

// 实现算法的一部分
class Warrior extends Character {
    @Override
    protected String getWeapon() {
        return "剑";
    }
}

// 实现算法的一部分
class Wizard extends Character {
    @Override
    protected String getWeapon() {
        return "魔杖";
    }
}

// 客户端
public class Game {
    public static void main(String[] args) {
        Character warrior = new Warrior();
        Character wizard = new Wizard();

        warrior.attack();
        wizard.attack();
    }
}
```

Iterator

将要遍历的集合类实现Iterable接口，表示这个集合类可以遍历，然后再实现Iterator类即可

```java
package basic_datastruct;

import java.util.Arrays;
import java.util.Iterator;

@SuppressWarnings({"all"})
public class DynamicArray<Item> implements Iterable<Item> {
    private int capacity;
    private Item[] array;
    private static final int DEFAULT_CAPACITY = 1;

    private void resize(int newCapacity) {
        array = Arrays.copyOf(array, newCapacity);
    }

    public DynamicArray() {
        array = (Item[])new Object[DynamicArray.DEFAULT_CAPACITY];
        capacity = 0;
    }

    public void add(Item item) {
        if (capacity == array.length) {
            resize(2 * array.length);
        }
        array[capacity++] = item;
    }

    public Item remove() {
        if (capacity > 0 && capacity <= array.length / 4) {
            resize(array.length / 2);
        }
        array[capacity - 1] = null;
        return array[--capacity];
    }

    public boolean isEmpty() {
        return capacity == 0;
    }

    public int size() {
        return capacity;
    }

    // 通过delegation来使用迭代器
    @Override
    public Iterator<Item> iterator() {
        return new DynamicArrayIterator();
    }

    // 自己的Iterator
    private class DynamicArrayIterator implements Iterator<Item> {
```

```
        private int n = capacity;

        @Override
        public boolean hasNext() {
            return n != 0;
        }

        @Override
        public Item next() {
            return array[--n];
        }

        @Override
        public void remove() {

        }
    }
}
```

visitor

访问者模式的核心是将操作封装在访问者对象中，从而使得您可以在不修改现有对象结构的情况下定义新的操作。访问者模式通过将操作和对象结构分离，从而使得您可以在不修改对象结构的情况下添加新的操作，从而使得代码更加灵活和可扩展。

在访问者模式中，有四个核心元素：抽象元素、具体元素、抽象访问者和具体访问者。抽象元素定义了一个accept()方法，该方法接受一个访问者对象，并调用访问者对象的visit()方法；具体元素扩展自抽象元素，并实现了accept()方法。抽象访问者定义了visit()方法的声明；具体访问者扩展自抽象访问者，并实现了visit()方法。

访问者模式的核心思想是将操作和对象结构分离，从而使得您可以在不修改对象结构的情况下添加新的操作。这使得访问者模式成为一种非常有用的设计模式，特别是在需要对现有对象结构进行复杂操作的情况下。

```
// 抽象元素
interface Shape {
    // 接受visitor
    void accept(Visitor visitor);
    double getArea();
}

// 具体元素
class Circle implements Shape {
    private double radius;

    public Circle(double radius) {
        this.radius = radius;
    }

    public void accept(Visitor visitor) {
        visitor.visitCircle(this);
    }

    public double getArea() {
        return Math.PI * radius * radius;
```

```
    }

    public double getPerimeter() {
        return 2 * Math.PI * radius;
    }
}

// 具体元素
class Rectangle implements Shape {
    private double width;
    private double height;

    public Rectangle(double width, double height) {
        this.width = width;
        this.height = height;
    }

    public void accept(Visitor visitor) {
        visitor.visitRectangle(this);
    }

    public double getArea() {
        return width * height;
    }

    public double getPerimeter() {
        return 2 * (width + height);
    }
}

// 抽象访问者
interface Visitor {
    void visitCircle(Circle circle);
    void visitRectangle(Rectangle rectangle);
}

// 具体访问者
class AreaVisitor implements Visitor {
    private double totalArea = 0;

    public void visitCircle(Circle circle) {
        totalArea += circle.getArea();
    }

    public void visitRectangle(Rectangle rectangle) {
        totalArea += rectangle.getArea();
    }

    public double getTotalArea() {
        return totalArea;
    }
}

// client
public class GraphicsApp {
    public static void main(String[] args) {
```

```
78          Shape[] shapes = new Shape[2];
79          shapes[0] = new Circle(5);
80          shapes[1] = new Rectangle(3, 4);
81
82          AreaVisitor areaVisitor = new AreaVisitor();
83
84          for (Shape shape : shapes) {
85              shape.accept(areaVisitor);
86          }
87
88          System.out.println("总面积：" + areaVisitor.getTotalArea());
89      }
90  }
```

Strategy vs visitor

- **Visitor: behavioral pattern**
- **Strategy: behavioral pattern**
- 二者都是通过delegation建立两个对象的动态联系
  - 但是Visitor强调是的外部定义某种对ADT的操作，该操作于ADT自身关系不大（只是访问ADT），故ADT内部只需要开放accept(visitor)即可，client通过它设定visitor操作并在外部调用。
  - 而Strategy则强调是对ADT内部某些要实现的功能的相应算法的灵活替换。这些算法是ADT功能的重要组成部分，只不过是delegate到外部strategy类而已。
- 区别：visitor是站在外部client的角度，灵活增加对ADT的各种不同操作（哪怕ADT没实现该操作），strategy则是站在内部ADT的角度，灵活变化对其内部功能的不同配置。

总结：Strategy主要适用于ADT内部函数的切换，Visitor主要是为了扩展ADT