- 设计模式（23种）
  - 行为型模式（11种）
    - 观察者模式
    - 模版方法模式
    - 策略模式
    - 责任链（职责链）模式
    - 访问者模式
    - 解释器模式
    - 命令模式
    - 迭代器模式
    - 状态模式
    - 备忘录模式
    - 中介者模式
  - 结构型模式（7种）
    - 适配器模式
    - 装饰模式
    - 享元模式
    - 组合模式
    - 桥接模式
    - 外观模式
    - 代理模式
  - 创建型模式（5种）

设计模式（23种）

行为型模式（11种）

观察者模式

模版方法模式

模式的定义与特点
模板方法（Template Method）模式的定义如下：定义一个操作中的算法骨架，而将算法的一些步骤延迟到子类中，使得子类可以不改变该算法结构的情况下重定义该算法的某些特定步骤。它是一种类行为型模式。

该模式的主要优点如下。

1. 它封装了不变部分，扩展可变部分。它把认为是不变部分的算法封装到父类中实现，而把可变部分算法由子类继承实现，便于子类继续扩展。
2. 它在父类中提取了公共的部分代码，便于代码复用。
3. 部分方法是由子类实现的，因此子类可以通过扩展方式增加相应的功能，符合开闭原则。

该模式的主要缺点如下。

1. 对每个不同的实现都需要定义一个子类，这会导致类的个数增加，系统更加庞大，设计也更加抽象，间接地增加了系统实现的复杂度。
2. 父类中的抽象方法由子类实现，子类执行的结果会影响父类的结果，这导致一种反向的控制结构，它提高了代码阅读的难度。
3. 由于继承关系自身的缺点，如果父类添加新的抽象方法，则所有子类都要改一遍。

```java
@Api(tags = "模板方法模式相关接口")
@Controller
public class TemplateMethodController {

 @Autowired
 HookConcreteTemplate hookConcreteTemplate;

 @PostMapping("/templatemethod")
 @ResponseBody
  public String templatemethod() {
  hookConcreteTemplate.templateMethod();
  return "success，见后台打印";
  }
 }
```

```java
/**
 * 含钩子方法的抽象类
 *
 * @author Sinu
 * @Date 2021/10/10
 */
public abstract class AbstractTemplate {

 //模板方法
  public void templateMethod() {
  abstractMethod1();
  hookMethod1();
  if (hookMethod2()) {
  specificMethod();
  }
  abstractMethod2();
  }

  //具体方法
  public void specificMethod() {
  System.out.println("抽象类中的具体方法被调用...");
  }

  //钩子方法1
  public void hookMethod1() {
```

```
    }

    //钩子方法2
    public boolean hookMethod2() {
    return true;
    }

    //抽象方法1
    public abstract void abstractMethod1();

    //抽象方法2
    public abstract void abstractMethod2();
}
```

```
@Component
public class HookConcreteTemplate extends AbstractTemplate {

    @Override
    public void abstractMethod1() {
    System.out.println("抽象方法1的实现被调用...");
    }

    @Override
    public void abstractMethod2() {
    System.out.println("抽象方法2的实现被调用...");
    }

    @Override
    public void hookMethod1() {
    System.out.println("钩子方法1被重写...");
    }

    @Override
    public boolean hookMethod2() {
    return false;
    }
}
```

参考：

[模板方法模式（模板方法设计模式）详解 (biancheng.net)]

策略模式

策略（Strategy）模式的定义：该模式定义了一系列算法，并将每个算法封装起来，使它们可以相互替换，且算法的变化不会影响使用算法的客户。策略模式属于对象行为模式，它通过对算法进行封装，把使用算法的责任和算法的实现分割开来，并委派给不同的对象对这些算法进行管理。

策略模式的主要优点如下。

1. 多重条件语句不易维护，而使用策略模式可以避免使用多重条件语句，如 if...else 语句、switch...case 语句。
2. 策略模式提供了一系列的可供重用的算法族，恰当使用继承可以把算法族的公共代码转移到父类里面，从而避免重复的代码。
3. 策略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或空间要求选择不同的。
4. 策略模式提供了对开闭原则的完美支持，可以在不修改原代码的情况下，灵活增加新算法。
5. 策略模式把算法的使用放到环境类中，而算法的实现移到具体策略类中，实现了二者的分离。

其主要缺点如下。

1. 客户端必须理解所有策略算法的区别，以便适时选择恰当的算法类。
2. 策略模式造成很多的策略类，增加维护难度。

```
@Api(tags = "策略模式相关接口")
@Controller
public class StrategyController {

 @Autowired
 FactoryForStrategy factoryForStrategy;

 @PostMapping("/strategy")
 @ResponseBody
  public String strategy(@RequestParam("key") String key) {
  String result;
  try {
  result = factoryForStrategy.getStrategy(key).doOperation();
```

```
 } catch (Exception e) {
 result = e.getMessage();
 }
 return result;
 }
}
```

```
/**
 * @author Sinu
 * @Date 2021/10/10
 */
public interface Strategy {
 String doOperation();
}
```

```
@Component("StrategyOne")
public class StrategyOne implements Strategy {
 @Override
 public String doOperation() {
 return "do one";
 }
}
```

```
@Component("StrategyTwo")
public class StrategyTwo implements Strategy {
 @Override
 public String doOperation() {
 return "do two";
 }
}
```

```
@Component("StrategyThree")
public class StrategyThree implements Strategy {
 @Override
 public String doOperation() {
 return "do three";
 }
}
```

```java
@Service
public class FactoryForStrategy {

 @Autowired
 Map<String, Strategy> strategys = new ConcurrentHashMap<>(3);

 public Strategy getStrategy(String component) throws Exception{
 Strategy strategy = strategys.get(component);
 if(strategy == null) {
  throw new RuntimeException("no strategy defined");
  }
  return strategy;
  }
}
```

Map<String, Strategy> strategys = new ConcurrentHashMap<>(3);
这里会自动注入，对应策略的实现类

参考：
策略模式（策略设计模式）详解 (biancheng.net)

责任链（职责链）模式
访问者模式
解释器模式
命令模式
迭代器模式
状态模式
备忘录模式
中介者模式

结构型模式（7种）

适配器模式
装饰模式
享元模式
组合模式
桥接模式
外观模式
代理模式

创建型模式（5种）

单例模式

建造者模式

抽象工厂模式

工厂模式

原型模式