软件架构与中间件

涂志莹
tzy_hit@hit.edu.cn

哈尔滨工业大学

软件架构与中间件

Software Architecture and Middleware

第2章
软件设计模式基础

第2章 软件设计模式基础

2.1 软件设计模式概述

2.2 常用软件设计模式

2.1 软件设计模式概述

- 软件设计模式简介
- 软件设计模式类型
- 软件设计模式原则

软件设计模式简介

软件设计模式的定义

- 软件设计模式 Software Design pattern
- **最佳的实践**：
  - 面向对象的软件开发人员的试验和错误的经验总结。
  - 软件开发过程中一般问题的解决方案。
- **代码编制工程化（软件工程的基石）：**
  - 重用代码
  - 让代码更容易被他人理解
  - 保证代码可靠性

每种软件设计模式都描述了一个在日常生产、生活中不断重复发生的问题，以及该问题的核心解决方案，都有相应的原理。

软件设计模式的用途

- **开发人员的共同平台**
  - 软件设计模式提供标准的术语系统，且具体到特定情景。
  - 例如，单例设计模式意味着使用单个对象，这样所有熟悉单例设计模式的开发人员都能使用单个对象，并且可以通过这种方式告诉对方，程序使用的是单例模式。
- **最佳的实践**
  - 软件设计模式提供了软件开发过程中面临的一般问题的最佳解决方案。
  - 软件设计模式有助于经验不足的开发人员通过一种简单快捷的方式来学习软件设计。

软件设计模式类型

软件设计模式的分类

- 《可复用的面向对象软件元素》(Elements of Reusable Object-Oriented Software) 一书中总结了25种模式，可以分为三大类：
  - 创建型模式（Creational Patterns）
  - 结构型模式（Structural Patterns）
  - 行为型模式（Behavioral Patterns）
- Java Web应用中还有：
  - J2EE 设计模式

创建型模式

- 一种在创建对象的同时**隐藏创建逻辑的方式**，而**不是使用 new** 运算符直接实例化对象。
- 程序在判断针对某个给定实例需要创建哪些对象时更加灵活。
- 包含：
  - **工厂模式（Factory Pattern）**
  - **抽象工厂模式（Abstract Factory Pattern）**
  - **单例模式（Singleton Pattern）**
  - 建造者模式（Builder Pattern）
  - 原型模式（Prototype Pattern）

结构型模式

- **关注类和对象的组合。**
- **继承的概念**被用来组合接口和定义组合对象获得新功能的方式。
- 包含：
  - **适配器模式（Adapter Pattern）**
  - **桥接模式（Bridge Pattern）**
  - 过滤器模式（Filter、Criteria Pattern）
  - 组合模式（Composite Pattern）
  - 装饰器模式（Decorator Pattern）
  - 外观模式（Facade Pattern）
  - 享元模式（Flyweight Pattern）
  - **代理模式（Proxy Pattern）**

行为型模式

- **关注对象之间的通信。**
- 包含：
  - 责任链模式（Chain of Responsibility Pattern）
  - 命令模式（Command Pattern）
  - 解释器模式（Interpreter Pattern）
  - 迭代器模式（Iterator Pattern）
  - **中介者模式（Mediator Pattern）**
  - 备忘录模式（Memento Pattern）
  - **观察者模式（Observer Pattern）**
  - 状态模式（State Pattern）
  - 空对象模式（Null Object Pattern）
  - 策略模式（Strategy Pattern）
  - 模板模式（Template Pattern）
  - **访问者模式（Visitor Pattern）**

J2EE 模式

- **关注表示层**。这些模式是由 Sun Java Center 鉴定的。
- 包含：
  - MVC 模式（MVC Pattern）
  - 业务代表模式（Business Delegate Pattern）
  - 组合实体模式（Composite Entity Pattern）
  - 数据访问对象模式（Data Access Object Pattern）
  - 前端控制器模式（Front Controller Pattern）
  - 拦截过滤器模式（Intercepting Filter Pattern）
  - 服务定位器模式（Service Locator Pattern）
  - 传输对象模式（Transfer Object Pattern）

模式间的关系

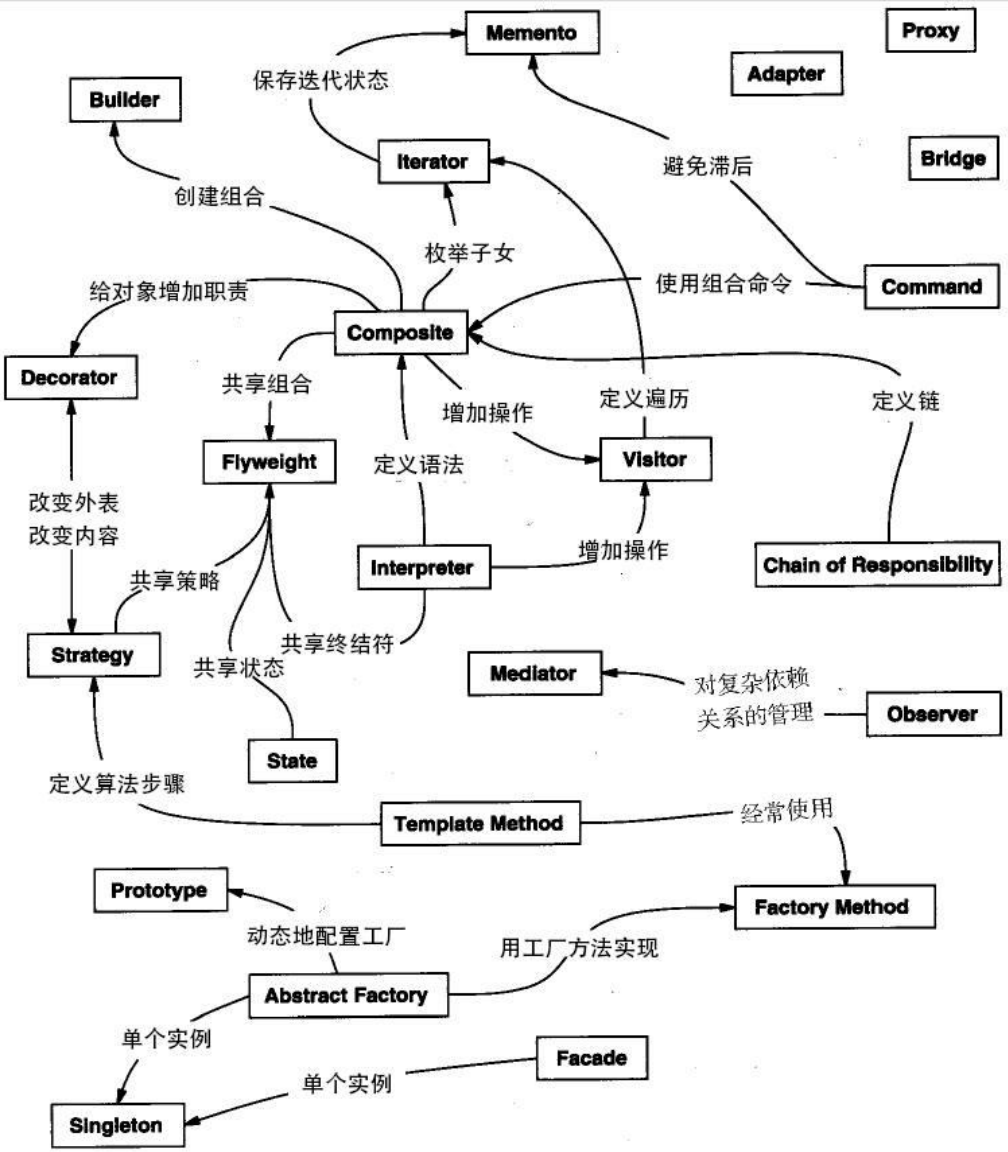


图　设计模式之间的关系

软件设计模式原则

设计模式原则

- **开闭原则（Open Close Principle）**
  - 对扩展开放，对修改关闭。
  - 在程序需要进行拓展的时候，不能去修改原有的代码，实现一个热插拔的效果。简言之，是为了使程序的扩展性好，易于维护和升级。
  - 想要达到这样的效果，需要使用接口和抽象类。

```
// 水果店
public class FruitShop {

  // 卖水果
  public void sellFruit(Fruit fruit) {
    if (fruit.fruit_type == 1) {
      sellApple(fruit);
    } else if (fruit.fruit_type == 2) {
      sellBanana(fruit);
    }
  }

  // 卖苹果
  public void sellApple(Fruit fruit) {
    System.out.println("卖出一斤苹果！");
  }

  // 卖香蕉
  public void sellBanana(Fruit fruit) {
    System.out.println("卖出一斤香蕉！");
  }
}
```

```
// 水果的基类
public class Fruit {

  int fruit_type;
}

// 苹果
public class Apple extends Fruit {

  Apple() {
    super.fruit_type = 1;
  }
}

// 香蕉
public class Banana extends Fruit {

  Banana() {
    super.fruit_type = 2;
  }
}
```

```
public class Watermelon extends Fruit {

  Watermelon() {
    super.fruit_type = 3;
  }
}
```

```
public void sellFruit(Fruit fruit) {
    if (fruit.fruit_type == 1) {
      sellApple(fruit);
    } else if (fruit.fruit_type == 2) {
      sellBanana(fruit);
    } else if(fruit.fruit_type == 3){
      sellWatermelon(fruit);
    }
}
```

```
public void sellWatermelon(Fruit fruit) {
    System.out.println("卖出一斤西瓜！");
}
```

设计模式原则

- **开闭原则（Open Close Principle）**
  - 对扩展开放，对修改关闭。
  - 在程序需要进行拓展的时候，不能去修改原有的代码，实现一个热插拔的效果。简言之，是为了使程序的扩展性好，易于维护和升级。
  - 想要达到这样的效果，需要使用接口和抽象类。

```
// 水果店
public class FruitShop {

  // 卖水果的方法
  public void sellFruit(Fruit fruit) {
    fruit.sell();
  }

}

// 水果的基类
public abstract class Fruit {

  int fruit_type;

  // 出售的方法
  public abstract void sell();
}
```

```
// 苹果
public class Apple extends Fruit {

  Apple() {
    super.fruit_type = 1;
  }

  @Override
  public void sell() {
    System.out.println("卖出一斤苹果！");
  }
}

// 香蕉
public class Banana extends Fruit {

  Banana() {
    super.fruit_type = 2;
  }

  @Override
  public void sell() {
    System.out.println("卖出一斤香蕉！");
  }
}
```

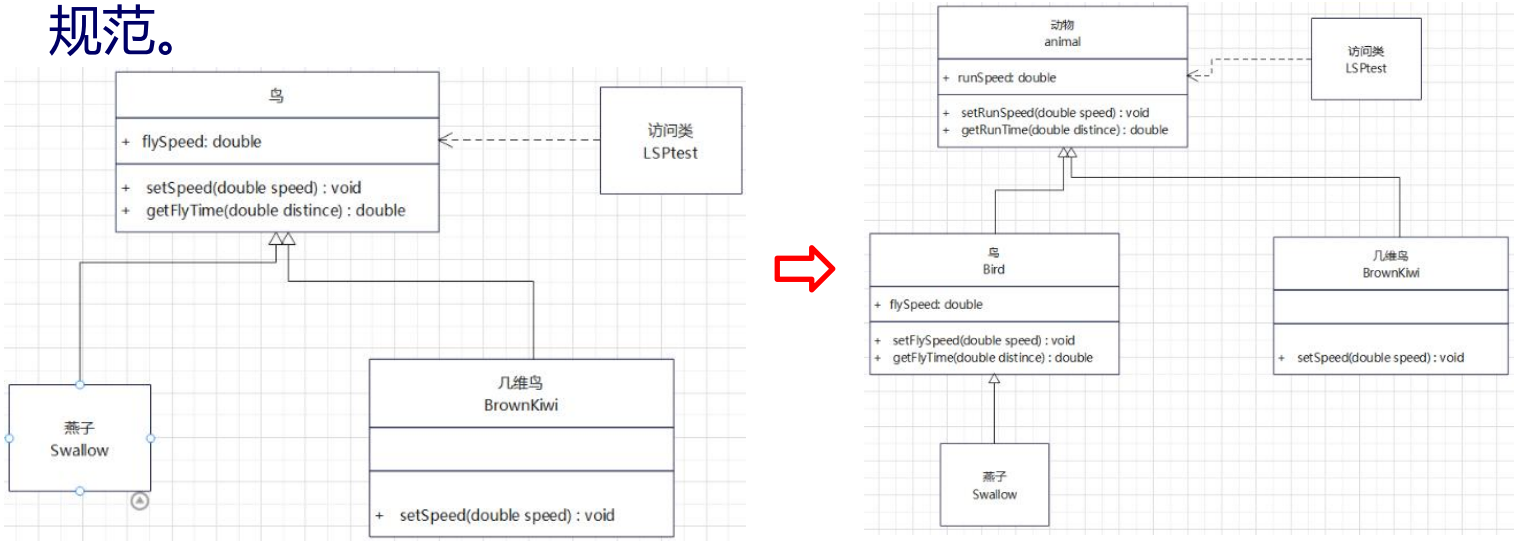
```
// 西瓜
public class Watermelon extends Fruit {

  Watermelon() {
    super.fruit_type = 3;
  }

  @Override
  public void sell() {
    System.out.println("卖出一斤西瓜！");
  }
}
```

设计模式原则

- **里氏代换原则（Liskov Substitution Principle）**
  - 任何基类可以出现的地方，子类一定可以出现。
  - LSP 是继承复用的基石，只有当派生类可以替换掉基类，且软件单位的功能不受到影响时，基类才能真正被复用，而派生类也能够在基类的基础上增加新的行为。
  - 里氏代换原则是对开闭原则的补充。
  - 实现开闭原则的关键步骤就是抽象化，而基类与子类的继承关系就是抽象化的具体实现，所以里氏代换原则是对实现抽象化的具体步骤的规范。



设计模式原则

- **依赖倒转原则（Dependence Inversion Principle）**
  - 这个原则是开闭原则的基础。
  - 具体内容：针对接口编程，依赖于抽象而不依赖于具体。

```java
public class DependencyInvertion {
    public static void main(String[] args) {
        Person person = new Person();
        person.receive(new Email());
    }
}

class Email {
    public String getInfo() {
        return "电子邮件信息: hello";
    }
}

//完成person接收消息的功能
class Person {
    public void receive(Email email) {
        System.out.println(email.getInfo());
    }
}
```

⇨

```java
public class DependencyInvertion {
    public static void main(String[] args) {
        //客户端无需改变
        Person person = new Person();
        person.receive(new Email());
        person.receive(new WeChat());
    }
}

interface IReceive {
    public String getInfo();
}

class Email implements IReceive{
    public String getInfo() {
        return "电子邮件信息: hello, Email";
    }
}

class WeChat implements IReceive {

    @Override
    public String getInfo() {
        return "微信信息: hello, Wechat";
    }
}

//完成person接收消息的功能
class Person {
    public void receive(IReceive receive) {
        System.out.println(receive.getInfo());
    }
}
```

设计模式原则

- **接口隔离原则（Interface Segregation Principle）**
  - 使用多个隔离的接口，比使用单个接口要好。
  - 客户端不应该依赖它 不需要的接口。降低类之间的耦合度。
  - 建立单一接口，尽量细化接口，接口中的方法尽量少。
  - 注意适度原则，一定要适度，过大的话会增加耦合性，而过小的话会增加复杂性和开发成本。

```java
public interface AnimalAction {
    void eat();
    void fly();
    void swim();
}
```

```java
public class Dog implements AnimalAction {
    @Override
    public void eat() {

    }

    @Override
    public void fly() {

    }

    @Override
    public void swim() {

    }
}
```

```java
public interface SwimAnimalAction {
    void swim();
}
```

```java
public interface EatAnimalAction {
    void eat();
}
```

```java
public interface FlyAnimalAction {
    void fly();
}
```

```java
public class Dog implements SwimAnimalAction, EatAnimalAction {
    @Override
    public void eat() {

    }

    @Override
    public void swim() {

    }
}
```

设计模式原则

- **迪米特法则，又称最少知道原则（Demeter Principle）**
  - 一个实体应当尽量少地与其他实体之间发生相互作用，使得系统功能模块相对独立。
  - 降低系统的耦合度，使类与类之间保持松耦合状态。

```java
public class Computer {
    public void saveCurrentTask() {
        // do something
    }

    public void closeService() {
        // do something
    }

    public void closeScreen() {
        // do something
    }

    public void closePower() {
        // do something
    }

    public void close() {
       saveCurrentTask();
       closeService();
       closeScreen();
       closePower();
    }
}
```

```java
public class Person {
    private Computer computer;

    public void clickCloseButton() {
        // 关闭计算机，正常来说你只要调用close()方法即可
        // 但是你发现Computer类所有的方法都是公开的，不知道怎么调用了
        computer.saveCurrentTask();
        computer.closePower();
        computer.close();

        //也可以是
        computer.closePower();

        //还可以是
        computer.close();
        computer.closePower();
    }
}
```

设计模式原则

- **迪米特法则，又称最少知道原则（Demeter Principle）**
  - 一个实体应当尽量少地与其他实体之间发生相互作用，使得系统功能模块相对独立。
  - 降低系统的耦合度，使类与类之间保持松耦合状态。

```java
public class Computer {
    private void saveCurrentTask() {
        // do something
    }

    private void closeService() {
        // do something
    }

    private void closeScreen() {
        // do something
    }

    private void closePower() {
        // do something
    }

    public void close() {
       saveCurrentTask();
       closeService();
       closeScreen();
       closePower();
    }
}
```

```java
public class Person {
    private Computer computer;

    public void clickCloseButton() {
        computer.close();
    }
}
```

设计模式原则

- **合成复用原则（Composite Reuse Principle）**
  - 复用类我们可以通过“继承”和“合成”两种方式来实现。
  - 尽量使用合成/聚合的方式，而不是使用继承。
    - 继承的优点：容易实现并且容易修改和扩展继承来的内容。
    - 继承的缺点：它最大的缺点就是增加了类之间的依赖，继承是属于“白箱”复用，父类对子类来说是透明的，这破坏了类的封装性。
    - 合成复用存在的缺点就是在系统中会存在较多的对象需要管理。

简单依赖

```
// 测试类
public class Test {

  public static void main(String[] args) {
    B b = new B();
    b.methodB(new A());
  }
}
// A类
class A {

  public void methodA() {
    System.out.println("A类的方法执行了。");
  }
}
// B类
class B {

  public void methodB(A a) {
    System.out.println("B类的方法执行了。");
    a.methodA();
  }
}
```

聚合

```
// 测试类
public class Test2 {

  public static void main(String[] args) {
    B b = new B();
    System.out.println("使用聚合的执行结果：");
    b.setA(new A());
    b.methodB();
  }
}
// A类
class A {

  public void methodA() {
    System.out.println("A类的方法执行了。");
  }
}
// B类
class B {

  private A a;

  public A getA() {
    return a;
  }
  public void setA(A a) {
    this.a = a;
  }

  public void methodB() {
    System.out.println("B类的方法执行了。");
    this.a.methodA();
  }
}
```

组合

```
public class Test3 {

  public static void main(String[] args) {
    B b = new B();
    System.out.println("使用组合的执行结果：");
    b.methodB();
  }
}

class A {

  public void methodA() {
    System.out.println("A类的方法执行了。");
  }
}

class B {

  private A a = new A();

  public void methodB() {
    System.out.println("B类的方法执行了。");
    this.a.methodA();
  }
}
```

不是不使用继承，而是尽量先考虑使用“合成”复用，其次在考虑使用“继承”复用。

第2章

软件设计模式基础

Thanks for listening

涂志莹
哈尔滨工业大学计算机学院
企业与服务计算研究中心