软件体系结构

《软件体系结构作业八》

学　号 22920212204396

姓　名 黄子安

2024 年 4 月 25 日

1、什么是 IoC（Inversion of Control）、 DIP（Dependency Inversion Principle）、Dependency Injection ？请举例说明实现方式。

IoC（Inversion of Control）是一种软件设计原则，它将控制权**从应用程序代码中转移到框架或容器中**，以实现松耦合和可扩展性。在 IoC 中，通常由框架或容器负责管理对象的创建、组装和生命周期，而不是由应用程序代码显式地控制，例如在 Spring 框架中就由框架完成 Bean 对象的创建与销毁，在需要的时候注入到对应的应用程序中，而非由应用程序本身通过 new 方法创建

DIP（Dependency Inversion Principle）是 SOLID 原则中的一部分，它要求高层模块不应该依赖于低层模块，两者都应该依赖于抽象。换句话说，模块之间的依赖关系应该是通过抽象而不是具体实现来建立的。从代码实现上来说就是高层模块不直接依赖于底层的模块，而是创建一个接口，高层模块调用这个接口，之后底层模块作为接口的实现，从而保证了满足 Liskov 可替换原则，实现解耦合和开闭原则

Dependency Injection（依赖注入）是实现 IoC 的一种方法，它通过将依赖对象的创建和管理从使用它们的类中分离出来，并通过外部注入的方式提供这些依赖对象。依赖注入有三种主要的实现方式：构造函数注入、Setter 方法注入和接口注入，在 Spring 框架中就可以在构造函数和 Setter 方法上添加@Autowired 注解来实现注入

下面通过不依赖 Spring 框架的 Java 代码更形象具体展示 **IoC 和 DI**

假设有一个类是 person，他们有一个行为是玩手机，根据传统的写法，如果他们想玩手机但是还没有手机的时候需要自己 new 一个出来，但这样子会有问题，比如手机这个类发生变化的时候，需要修改 person 的代码，例如构造函数的参数发生改变等都会导致代码改变，假设由很多个地方都 new 一个 phone 对象，则需要大量修改代码

```
public class Person {
    private Phone myPhone;
    public Person(Phone phone)
    {
        this.myPhone = phone;
    }

    public void playPhone() {
//        this.myPhone = new Phone("iphone");
       System.out.println("play:"+myPhone.getName());
    }
}
```

```
public class Phone {
    private String name;
    public Phone(String name) {
        this.name = name;
    }

    public String getName()
    {
        return this.name;
    }
}
```

根据 IoC 和 DI 的思想，现在把控制权转交给别人，在 main 函数中先创建好 phone 对象，之后通过构造函数注入给 person 对象，这样 person 对象只管玩手机即可，具体玩的是什么手机不需要自己创建，直接由外界送过来即可

```
public class Main {
    public static void main(String[] args) {
        Phone phone1 = new Phone("iphone");
        Person  person1 = new Person(phone1);
        person1.playPhone();

        Phone phone2 = new Phone("HUAWEI");
        Person  person2 = new Person(phone2);
        person2.playPhone();
    }
}
```

运行结果如下：

```
D:\Java17\bin\java.exe "-javaagent:D:\IdeaU\IntelliJ IDEA 2023.1.3\lib\idea_rt.jar=51540:D:\IdeaU\IntelliJ IDEA 202
play:iphone
play:HUA WEI

Process finished with exit code 0
```

这个思想除了解耦合外也将在单元测试中发挥巨大作用，比如编写好了 person 代码需要测试是不是会正常玩手机，但是此时不确定 phone 本身是否代码正确，就可以通过切片测试的方法在待测试的 person 对象中注入一个假的 phone，这个假的 phone 是模拟出来的，对于给定测试用例的输出都是自己直接编写，保证 phone 本身是对的，从而实现对 person 单元测试

Spring 框架也是同理，只是对于 Bean 对象的创建不是放在 Main 函数中，而是通过 XML 或者注解等定义，在应用程序启动时，先初始化 Spring 容器，并加载配置文件或配置类，之后由框架通过工厂模式进行创建，应用程序需要 Bean 对象的时候由容器提供，实现了 Spring 容器负责管理 Bean 的创建和依赖关系，将控制权从应用程序代码中转移到了 Spring 容器中，实现了松耦合、可维护和可扩展的应用程序设计

再举例说明**控制反转**，先举例一个不使用控制反转的例子，这段代码高层订单部分直接依赖于低层的支付部分，在运行后确实可以完成任务，但是 OrderService 和 AlipayPaymentService 之间的耦合度很高，如果要更换支付方式，需要修改 OrderService 的代码，违反了开闭原则，此外单元测试时也难以进行模块的替换和独立测试。

```java
public class Order {
    public double getTotalAmount() {
        return 100;
    }
}

// 低层模块 AlipayPaymentService
public class AlipayPaymentService {
    public void processPayment(double amount) {
        System.out.println("调用支付宝支付接口完成支付");
    }
}

// 高层模块 OrderService 依赖于低层模块 AlipayPaymentService
public class OrderService {
    private AlipayPaymentService paymentService;

    public OrderService() {
        this.paymentService = new AlipayPaymentService();
    }

    public void processOrder(Order order) {
        this.paymentService.processPayment(order.getTotalAmount());// 处理订单逻辑
    }
}

public class Main {
    public static void main(String[] args) {
        OrderService orderService = new OrderService();
        Order order = new Order();
        orderService.processOrder(order);
    }
}
```

运行结果：

```
D:\Java17\bin\java.exe "-javaagent:D:\IdeaU\IntelliJ IDEA 2023.1.3\lib\id
调用支付宝支付接口完成支付

Process finished with exit code 0
```

之后使用依赖倒置修改代码，定义一个抽象的接口，订单部分不再直接依赖于具体的支付宝支付，还是依赖于一个抽象的支付方式，之后再去实现这个抽象的接口，这样就可以很方便的进行支付方式的切换，同时这里也体现了刚才的IoC和DI，对于 OrderService 来说，其中的支付对象不再是自己创建的，而是由外部创建注入进来，大大减少了代码耦合和增加了模块化程度

```
public interface PaymentService {
    void processPayment(double amount);
}

public class WechatPaymentService implements PaymentService {
    public void processPayment(double amount) {
        System.out.println("调用微信支付接口完成支付");
    }
}

public class AlipayPaymentService implements PaymentService{
    public void processPayment(double amount) {
        System.out.println("调用支付宝支付接口完成支付");
    }
}
public class OrderService {
    private PaymentService paymentService;

    public OrderService(PaymentService paymentService) {
        this.paymentService = paymentService;
    }

    public void processOrder(Order order) {
        // 处理订单逻辑
        this.paymentService.processPayment(order.getTotalAmount());
    }
}

public class Main {
    public static void main(String[] args) {
        PaymentService alipay = new AlipayPaymentService();
        WechatPaymentService wechat = new WechatPaymentService();

        OrderService orderService1 = new OrderService(alipay);
        OrderService orderService2 = new OrderService(wechat);

        Order order = new Order();
        orderService1.processOrder(order);
        orderService2.processOrder(order);
    }
}
```

运行结果如下，现在可以很方便的想使用何种渠道支付就使用何种渠道，也可以轻松地扩展新的支付方式，例如添加银联支付等，而无需修改 OrderService 的代码，符合开闭原则，也更容易进行单元测试，可以通过模拟 PaymentService 接口来测试 OrderService 的逻辑

```
D:\Java17\bin\java.exe "-javaagent:D:\IdeaU\IntelliJ IDEA 2023.1.3\lib\idea_rt.jar=52019:D:\Id
调用支付宝支付接口完成支付
调用微信支付接口完成支付

Process finished with exit code 0
```

从类图上来看可以很直观发现高层模块不再直接依赖于低层模块了，使用一个接口进行承上启下，实现封装和多态