SA第七次作业

HomeWork7

什么是双向适配器?

双向适配器是一种设计模式，用于在两个不兼容的接口之间进行通信和转换。它允许两个系统、组件或对象之间进行双向通信，同时在它们之间执行必要的转换或适配，以确保彼此之间的交互能够顺利进行。即Adaptee类可以通过适配器调用Target类中的方法，Target类也可以通过适配器调用Adaptee类的方法。

实现方式

Adaptee接口

```
public interface Adaptee {
    void specificRequest();
}
```

AdapteeImpl（Adaptee的实现）

```
public class AdapteeImpl implements Adaptee{
    @Override
    public void specificRequest() {
        System.out.println("Adaptee方法");
    }
}
```

Target类

```
public interface Target {
    void request();
}
```

TargetImpl（Target的实现）

```
public class TargetImpl implements Target{
    @Override
    public void request() {
        System.out.println("Target方法");
    }
}
```

Adapter适配器类，同时实现Target和Adaptee，使得Adaptee类可以通过Adapter调用Target类中的方法，Target类也可以通过Adapter调用Adaptee类的方法。

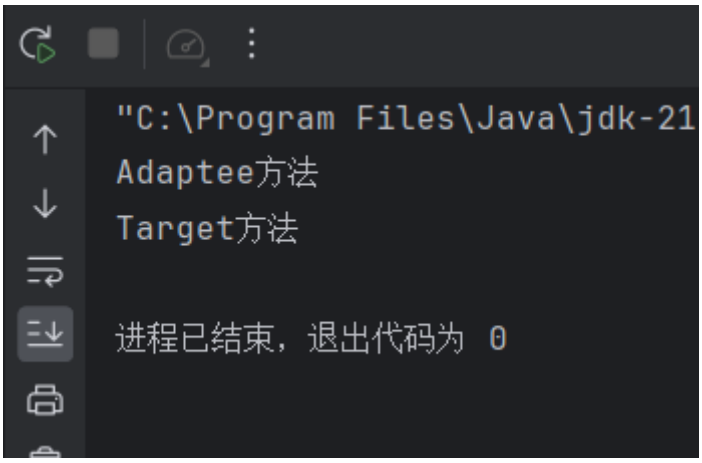
```
public class Adapter implements Adaptee,Target{
    private Target target = new TargetImpl();
    private Adaptee adaptee = new AdapteeImpl();
    @Override
    public void specificRequest() {
        //适配者方法调用Target的方法
        target.request();
    }

    @Override
    public void request() {
        //Target的方法调用适配者方法
        adaptee.specificRequest();
    }
}
```

Main函数

```
public class Main {
    public static void main(String[] args) {
        Adapter adapter = new Adapter();
        adapter.request();
        adapter.specificRequest();
    }
}
```

运行结果



HomeWork8

什么是IoC？

控制反转（Inversion of Control），是面向对象编程中的一种设计原则，可以用来减低计算机代码之间的耦合度。其中最常见的方式叫做**依赖注入**（Dependency Injection，简称DI），还有一种方式叫“依赖查找”（Dependency Lookup）。通过控制反转，对象在被创建的时候，由一个调控系统内所有对象的外界实体将其所依赖的对象的引用传递给它。也可以说，依赖被注入到对象中。

什么是DI？

依赖注入（Dependency Injection, DI）是一种设计模式和编程技术，目的是减少代码中对象之间的耦合度，提高代码的可维护性、灵活性和可测试性。它通过将对象的依赖关系**由外部容器管理和注入**，实现控制反转（Inversion of Control, IoC），其中对象的创建、组装和依赖关系的管理由外部容器负责。

什么是DIP？

DIP（Dependency Inversion Principle），即依赖倒置原则：高层模块不应该依赖低层模块，两者都应该依赖其抽象；抽象不应该依赖细节，细节应该依赖抽象。

实现方式

Engine接口

```
public interface Engine {
    void start();
}
```

ElectricEngine类（实现Engine接口）

```
public class ElectricEngine implements Engine{
    @Override
    public void start() {
        System.out.println("电力引擎启动。");
    }
}
```

GasEngine类（实现Engine接口）

```
public class GasEngine implements Engine{
    @Override
    public void start() {
        System.out.println("燃气发动机启动。");
    }
}
```

Car类

Car类依赖于Engine这一抽象，而不是依赖于具体的某种特定引擎实现类，通过构造函数注入依赖，Car 对象可以使用任何实现了 Engine 接口的类，而不需要修改 Car 类的代码。

```
public class Car {
    private Engine engine;

    // 通过构造函数注入依赖
    public Car(Engine engine) {
        this.engine = engine;
    }

    public void start() {
        // 依赖抽象，而不是具体实现
        engine.start();
    }
}
```

Main函数

```
public class Main {
    public static void main(String[] args) {
        // 创建汽车对象，传入具体的引擎实现
        Car electricCar = new Car(new ElectricEngine());
        // 启动汽车
        electricCar.start();

        // 创建汽车对象，传入具体的引擎实现
        Car gasCar = new Car(new GasEngine());
        // 启动汽车
        gasCar.start();
    }
}
```

运行结果