优秀借鉴

1. 黑马程序员Java设计模式详解-适配器模式概述
2. 适配器设计模式（封装器模式）
3. 一文彻底弄懂适配器模式(Adapter Pattern)
4. 《深入设计模式》--亚历山大·什韦茨

1、简介

适配器模式是一种常用的结构型设计模式，**核心思想**是将现有的接口转换为客户端所期望的接口。它允许通过将一个接口转换为另一个接口，将不兼容的类或对象组合在一起。这种模式通常用于集成现有系统或库中不兼容的组件。

在软件开发中，我们经常会遇到由不同的团队或不同的供应商编写的代码、服务或库，这些组件可能使用不同的协议、数据格式或接口定义，因此无法直接集成在一起。为了解决这个问题，我们可以使用适配器模式来创建一个适配器，它可以将这些不兼容的组件转换为一个统一的接口，从而实现它们之间的互操作性。

2、结构

当我们使用适配器模式时，通常会涉及到以下三个角色：

1. **目标接口**（Target）：该角色是所需的客户端接口，也就是客户端希望使用的接口。在适配器模式中，我们需要设计一个新的目标接口来满足客户端的需求。
2. **适配器**（Adapter）：该角色是适配器模式的核心，其作用是将不兼容的接口转换为目标接口。适配器可以通过继承或组合等方式实现。
3. **源接口**（Adaptee）：该角色是需要被适配的现有接口，它与目标接口不兼容，无法直接使用。在适配器模式中，我们需要将源接口适配成目标接口，以便客户端能够使用。

3、实现方式

3.1、案例引入

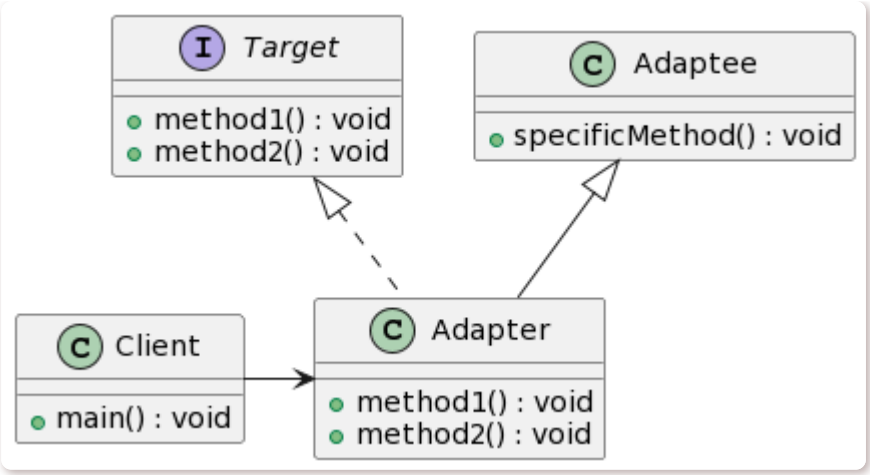
不知道有没有尊贵的Mac用户在想要外接显示屏时，却苦于电脑只有 type-c 口而没有视频口，这个时候就需要一个 type-c 转 HDMI 的转接器了，毕竟我们总不能把电脑拆开自己加上一个 HDMI 接口对吧，要是不小心把电脑搞坏了还得花大成本去维修。

对于上述场景中其实就用到了一个适配器模式，对应到适配器中的三个角色分别如下：

1. **目标接口**：对应的是 HDMI 线所需要的 **HDMI 接口**，也就是我们希望使用的接口；
2. **适配器**：对应的是**转接器**，作用就是将不兼容的 **type-c 接口**转换成目标接口 **HDMI 接口**；
3. **源接口**：对应的是 **type-c 接口**，就是电脑现有的接口，与我们希望的目标接口 **HDMI 接口**不兼容，无法直接使用。

3.2、类适配器

类适配器通过**继承**来适配两个不兼容的接口。



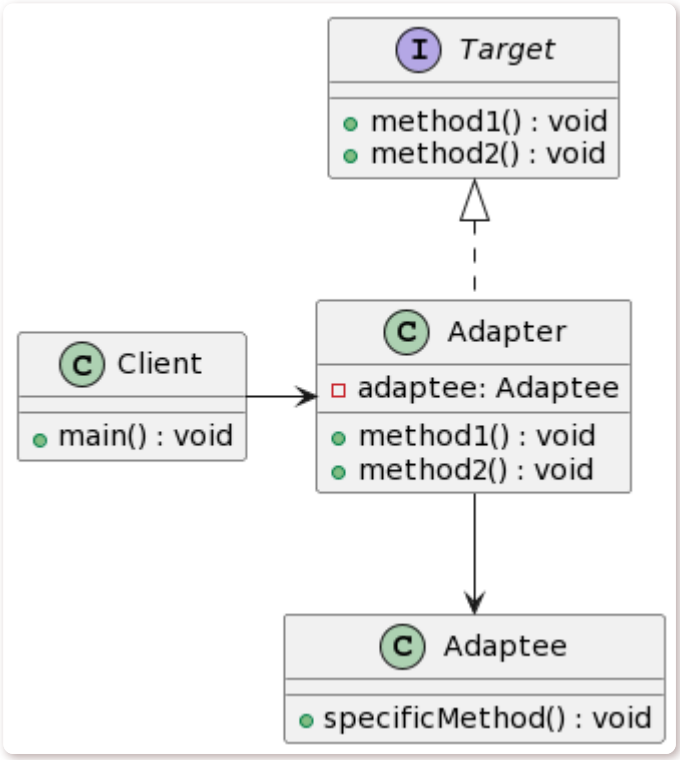
下面是使用类适配器将Type-C接口转换为HDMI接口的代码实现：

```
// Type-C 接口
public interface TypeC {
    void sendDisplay(String content);
}

// HDMI 接口
public interface HDMI {
    void display(String content);
}

// Type-C 到 HDMI 适配器
public class TypeCToHDMIAdapter extends MacBook implements TypeC, HDMI {
    @Override
    public void sendDisplay(String content) {
        System.out.println("Type-C 转 HDMI 适配器: " + content);
        super.sendDisplay(content);
    }

    @Override
    public void display(String content) {
        System.out.println("Type-C 转 HDMI 适配器: " + content);
        super.display(content);
    }
}

// 笔记本电脑类
public class MacBook {
    public void sendDisplay(String content) {
        System.out.println("笔记本电脑发送显示信号: " + content);
    }

    public void display(String content) {
        System.out.println("在笔记本电脑屏幕上显示: " + content);
    }
```

```
}
```

3.3、对象适配器

对象适配器通过**组合**另一个类来适配两个不兼容的接口。



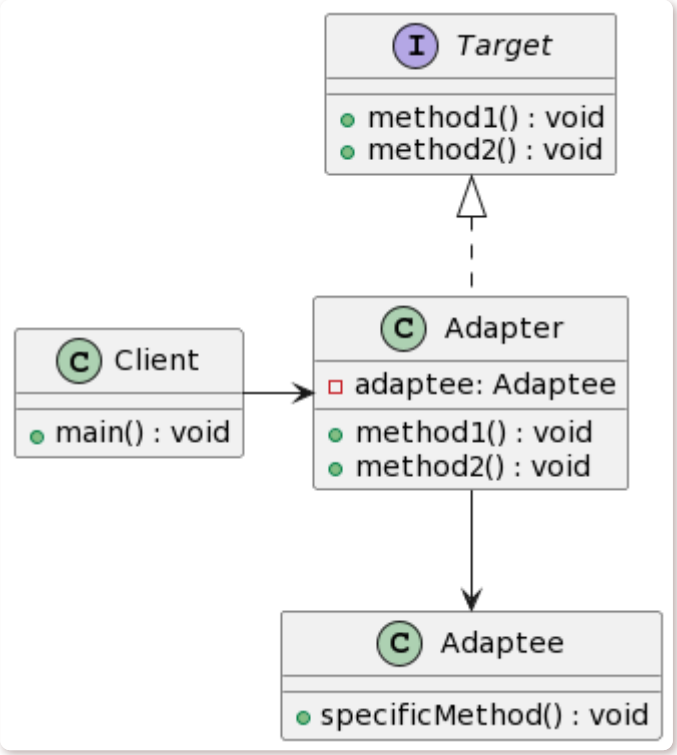
下面是使用对象适配器将 Type-C 接口转换为 HDMI 接口的代码实现：

```java
// Type-C 接口
public interface TypeC {
    void sendDisplay(String content);
}

// HDMI 接口
public interface HDMI {
    void display(String content);
}

// Type-C 到 HDMI 适配器
public class TypeCToHDMIAdapter implements TypeC {
    private final HDMI hdmi;

    public TypeCToHDMIAdapter(HDMI hdmi) {
        this.hdmi = hdmi;
    }

    @Override
    public void sendDisplay(String content) {
        System.out.println("Type-C 转 HDMI 适配器: " + content);
```

```
        hdmi.display(content);
    }
}

// HDMI 接口实现类
public class HDMIImpl implements HDMI {
    @Override
    public void display(String content) {
        System.out.println("在 HDMI 屏幕上显示: " + content);
    }
}
```

3.4、接口适配器

接口适配器通过一个抽象类来适配多个不兼容的接口。在 Java 中，可以使用抽象类和默认方法来实现适配器。



下面是使用接口适配器将 Type-C 接口转换为 HDMI 接口的代码实现：

```
// Type-C 接口
public interface TypeC {
    void sendDisplay(String content);
}

// HDMI 接口
public interface HDMI {
    void display(String content);
}

// 抽象适配器类
```

```java
public abstract class TypeCToHDMIAdapter implements TypeC, HDMI {
    @Override
    public void sendDisplay(String content) {
        System.out.println("Type-C 转 HDMI 适配器: " + content);
    }

    @Override
    public void display(String content) {
        System.out.println("Type-C 转 HDMI 适配器: " + content);
    }
}

// 笔记本电脑类
public class Laptop extends TypeCToHDMIAdapter {
    @Override
    public void sendDisplay(String content) {
        System.out.println("笔记本电脑发送显示信号: " + content);
        super.sendDisplay(content);
    }

    @Override
    public void display(String content) {
        System.out.println("在笔记本电脑屏幕上显示: " + content);
        super.display(content);
    }
}
```

4、区别对比

类适配器、对象适配器、接口适配器都是适配器模式的实现方式，它们的目的是将一个类或接口转换成另一个类或接口，以满足不同的业务需求。它们之间的联系和区别如下：

1. **类适配器**：类适配器通过**继承**待适配类和实现目标接口的方式，来实现对待适配类的适配。具体来说，在适配器中包含了待适配类的实例，并实现了目标接口的方法，以便客户端调用。这种方式可以在不改变已有代码的情况下进行适配，但只能适配单个待适配类。
2. **对象适配器**：对象适配器通过**组合**待适配类的实例和实现目标接口的适配器类的方式，来实现对待适配类的适配。具体来说，在适配器中包含了待适配类的实例，并通过实现目标接口的方式，将待适配类的方法委托给适配器来实现。这种方式可以适配多个待适配类，而且更加灵活，因为可以在运行时动态设置待适配类的实例。
3. **接口适配器**：接口适配器通过定义一个**抽象适配器类**，实现目标接口的所有方法，并将它们设置成空方法。待适配类只需要实现需要的方法即可，避免了实现不必要的方法，也使得适配器更加灵活。

特点	类适配器	对象适配器	接口适配器
实现方式	继承	组合	抽象类

特点	类适配器	对象适配器	接口适配器
适配范围	单个类	多个类	多个方法
灵活性	低	高	中等
对待适配类的影响	有	有	小

5、适配者模式优缺点

适配器模式是一种常用的设计模式，它可以将一个类或接口转换成另一个类或接口，以满足不同的业务需求。适配器模式有以下优缺点：

优点：

1. **提高代码复用性**：适配器模式可以重用已有的代码，减少代码量；
2. **提高系统的灵活性**：适配器模式可以使得系统更加灵活，易于扩展和维护；
3. **降低耦合度**：适配器模式可以将不同的模块之间解耦，使得各个模块之间的依赖关系更加简单明了；
4. **可以适配多个类或接口**：不同的适配器实现方式可以适配多个类或接口，提高代码的可复用性。

缺点：

1. **增加代码复杂性**：适配器模式需要增加新的适配器类或方法，会增加代码的复杂性；
2. **可能会造成性能损失**：适配器模式可能会引入额外的开销，例如对象适配器需要组合待适配类的实例对象；
3. **不易理解**：适配器模式可能会使代码结构变得复杂，不易于阅读和理解。

优点	缺点
提高代码复用性	增加代码复杂性
提高系统的灵活性	可能会造成性能损失
降低耦合度	不易理解
可以适配多个类或接口	

6、应用场景

适配器模式是一种常用的设计模式，主要应用于以下场景：

1. 处理旧接口与新接口的**兼容性问题**：当系统中的某个组件需要调用另一个组件的接口时，如果这两个组件的接口不兼容，可以使用适配器模式将旧接口转换成新接口；
2. **重用**已有的代码：适配器模式可以重用现有的代码，减少代码量，提高代码的可复用性；
3. **构建抽象接口**：适配器模式可以将多个类或接口适配成一个抽象接口，使得客户端只需要针对抽象接口编程，而不需要关注具体的实现细节；

4. **隐藏**不必要的接口：适配器模式可以隐藏一些不必要的接口，避免客户端直接访问实现类的方法，提高代码的安全和稳定性；
5. **适配不同的数据格式**：适配器模式可以适配不同的数据格式，例如将 XML 数据转换成 JSON 格式。