文章已收录我的仓库：Java学习笔记与免费书籍分享

设计意图

适配器模式（Adapter Pattern）是作为两个不兼容的接口之间的桥梁。这种类型的设计模式属于结构型模式，它结合了两个独立接口的功能。

在某些时候，客户期望获得某种功能接口但现有的接口无法满足客户的需求，例如美国的正常供电电压为110V，一个中国人带了一款中国制造电器去美国，这个电器必须要在220V电压下才能充电使用。这种情况下，客户(中国人)的期望接口是`有一个220V的电压为电器充电`，但实际的接口是`仅有一个110V的电压供电器充电`，这种情况下就需要采用一根电压转换器(适配器)使得110V的电压能够转换为220V的电压，供客户使用。

将一个类的接口转换成客户希望的另外一个接口，这就是适配器需要做的事情，适配器模式使得原本由于接口不兼容而不能一起工作的那些类可以一起工作。

适用条件

- 系统需要使用现有的类，而此类的接口不符合系统的需要(核心需求)。
- 想要建立一个可以重复使用的适配器类，用于与一些彼此之间没有太大关联的一些类，包括一些可能在将来引进的类一起工作，这些源类不一定有一致的接口，但通过适配器使得它们都具有一致的接口。
- 通过接口转换，将一个类插入另一个类系中。（比如老虎和飞禽，现在多了一个飞虎，在不增加实体的需求下，增加一个适配器，在里面包容一个虎对象，实现飞的接口。）

设计

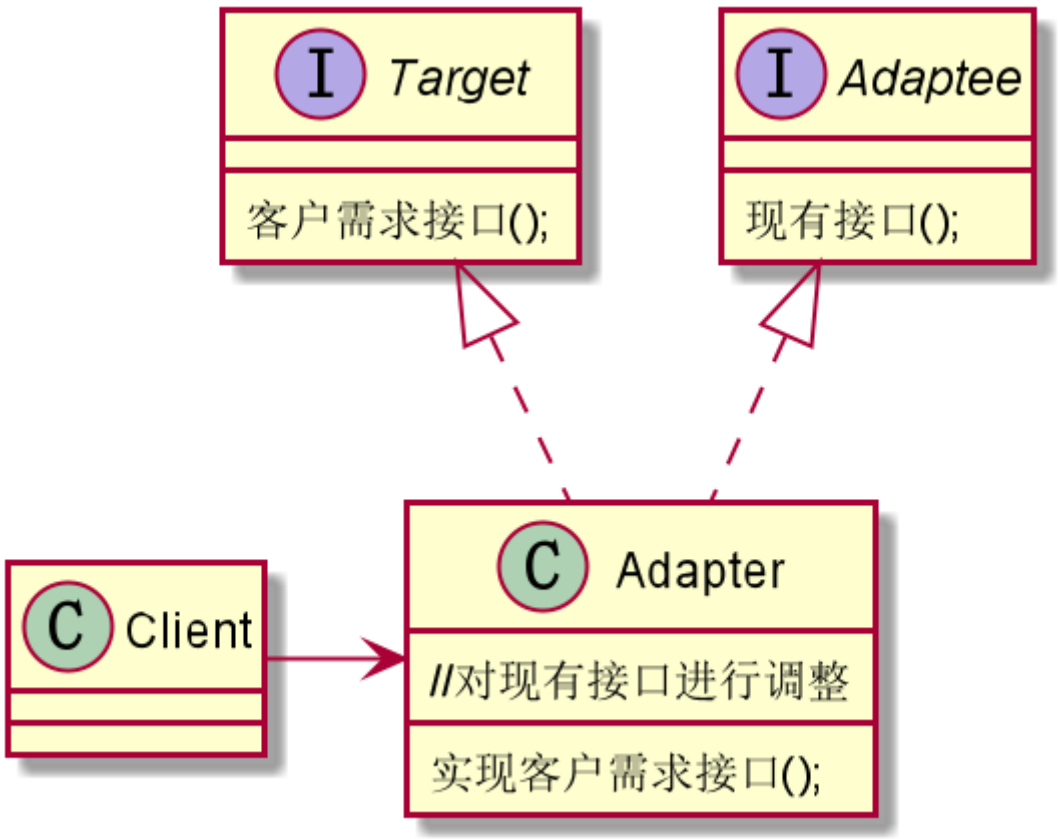
通常有两种方式实现适配器模式，一种是类适配器，类适配器目前已不太使用，另一种实现方式是对象适配器，通常情况下采用对象适配器会使得代码更易扩展与维护。

不管采用何种方式，其基本的实现思想都是：对现有接口的实现类进行扩展，使其实现客户期望的目标接口。

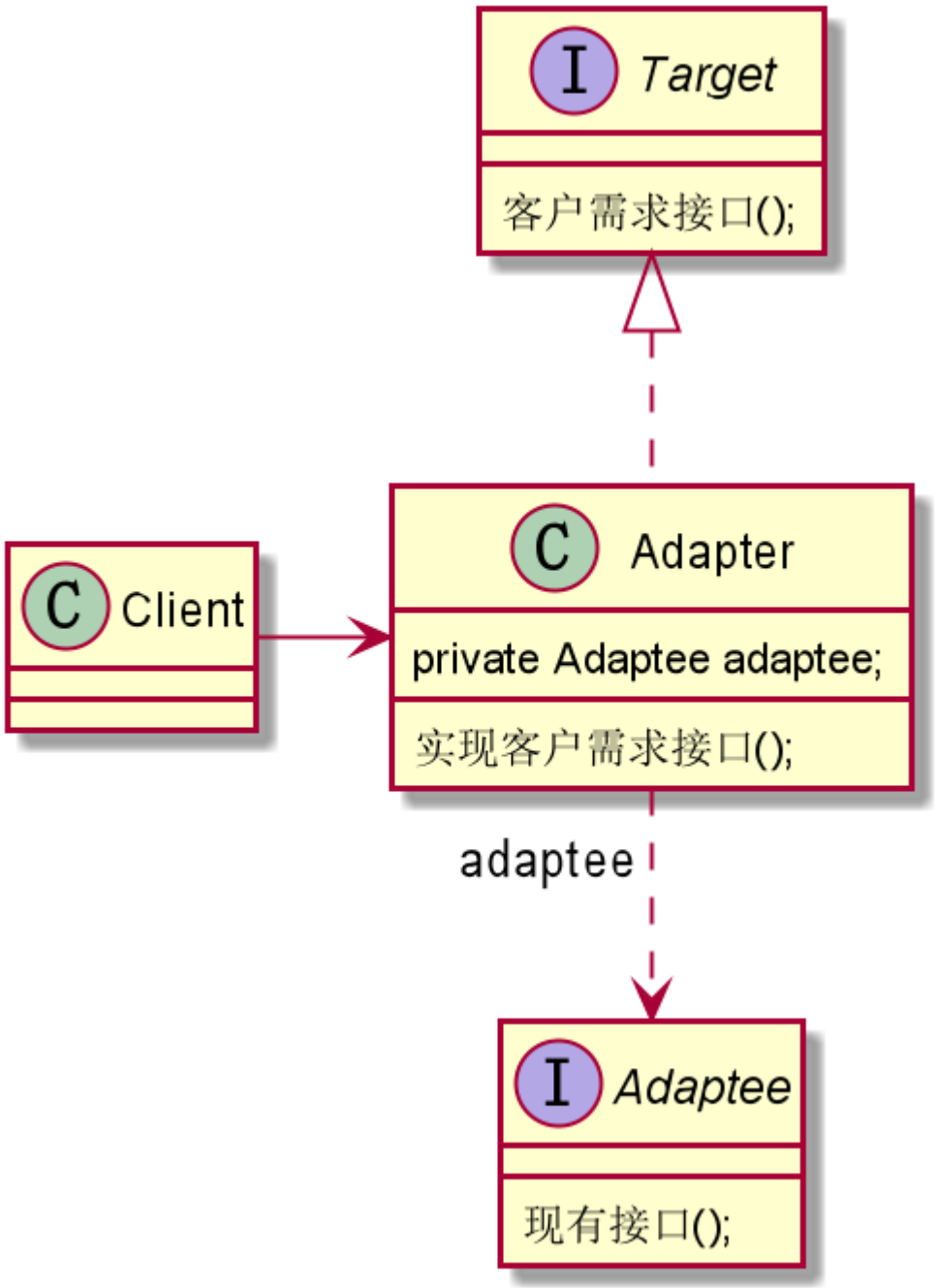
类适配器通过继承现有接口类并实现目标接口，这样的话会使得现有接口类完全对适配器暴露，使得适配器具有现有接口类的全部功能，破坏了封装性。此外从逻辑上来说，这也是不符合常理的，适配器要做的是扩展现有接口类的功能而不是替代，类适配器只有在特定条件下会被使用。

对象适配器持有现有接口类一个实例，并扩展其功能，实现目标接口。这是推荐的方式，优先采用组合而不是继承，会使得代码更利于维护。此外，这也是非常符合常理的——“给我一根线，让我来给他加长到5m，我并不需要知道这跟线是什么组成的，因为我的工作就是让线加长到5m”——我们扩展了相应功能而并不关心其具体实现。

类适配器结构图：



对象适配器结构图：



- Target：客户期望获得的功能接口(220V电压供电)。
- Cilent：客户，期望访问Target接口(客户期望能有220V电压)。
- Adaptee：现有接口，这个接口需要被适配(现有110V电压供电，需要被适配至220V)。
- Adapter：适配器类，适配现有接口使其符合客户需求接口(适配110V电压，使其变为220V电压)。

在适配器模式中，Cilent调用Adapter以获得相应功能，Adapter扩展Adaptee以实现对应功能。

代码示例

类适配器：

```java
//客户期望的接口——220V的电压充电
interface Target {
    void chargeBy220V();
}

//现有接口——只能通过110V电压充电
interface Adaptee {
    void chargeBy110V();
}

//现有接口的具体实现类，美国供电器——通过110V电压供电
class americanCharger implements Adaptee {
    @Override
    public void chargeBy110V() {
        System.out.println("美国供电器，只为你服务，正在通过110V电压为您充电");
    }
}

//类适配器，通过继承现有接口来完成对现有接口的扩展
class Adapter extends americanCharger implements Target {
    @Override
    public void chargeBy220V() {
        super.chargeBy110V();//现有功能
        System.out.println("再加110V，达到220V，冲鸭！");//对现有功能扩展
    }
}

//测试类
public class Test {
    public static void main(String[] args) throws FileNotFoundException {
        //类适配器使得代码逻辑混乱
        //这种情况下仿佛Adapter是一种110V的美国供电器可以直接使用不需要其他信息
        //具体可以和对象适配器对比以下
        new Adapter().chargeBy220V();
    }
}

//输出
/*
美国供电器，只为你服务，正在通过110V电压为您充电
再加110V，达到220V，冲鸭！
*/
```

对象适配器：

```java
//客户期望的接口——220V的电压充电
interface Target {
    void chargeBy220V();
}

//现有接口——只能通过110V电压充电
interface Adaptee {
```

```
    void chargeBy110V();
}

//现有接口的具体实现类，美国供电器——通过110V电压供电
class americanCharger implements Adaptee {
    @Override
    public void chargeBy110V() {
        System.out.println("美国供电器，只为你服务，正在通过110V电压为您充电");
    }
}

//类适配器，通过继承现有接口来完成对现有接口的扩展，使得能够110V供电
class Adapter implements Target {
    Adaptee adaptee;//持有现有接口具体实现对象的引用

    public Adapter(Adaptee adaptee) {
        this.adaptee = adaptee;
    }

    @Override
    public void chargeBy220V() {
        adaptee.chargeBy110V();//该对象的现有功能
        System.out.println("再加110V，达到220V，冲鸭！");//对现有功能扩展
    }
}

//测试类
public class Test {
    public static void main(String[] args) throws FileNotFoundException {
        //现在我们有一个美国110V供电站，但我们无法使用
        Adaptee adaptee = new americanCharger();

        //我们将这个供电器交给适配器，适配器转换为220V供电器
        Adapter adapter = new Adapter(adaptee);

        //接下来我们通过适配器充电就好了
        adapter.chargeBy220V();
    }
}
//输出同上
```

对象适配器采用组合的方式实现对现有接口的扩展以达到客户期望的接口。

让我们来看JavaIO流中的一个实例：

```
FileInputStream fis = new FileInputStream("qe");
InputStreamReader isrAdapter = new InputStreamReader(fis);
BufferedReader bf = new BufferedReader(isrAdapter);
```

BufferedReader(此处为客户)需要读取文件字符流进行工作，读取文件字符流就是客户的需求部分，但是根据现有的接口，想要读取文件就只能读取字节流，FileInputStream

就是现有接口的一个具体实现类，为了满足客户的需求，我们要对现有的接口进行适配，InputStreamReader就是一个适配器，它持有一个现有接口类的实例，通过这个实例读取文件字节流并将其扩展为字符流以满足客户的需求，这是标准的对象适配器模式。如果仔细研究源码，发现JavaIO库将适配器定义为抽象的，并由具体的适配器继承该抽象适配器，如这里的InputStreamReader就是具体的适配器之一。

如果实现适配有多种方式的话，我们可以将适配器类Adapter声明为抽象类，并由子类扩展它:

```
//客户期望的接口——220V的电压充电
interface Target {
    void chargeBy220V();
}

//现有接口——只能通过110V电压充电
interface Adaptee {
    void chargeBy110V();
}

//现有接口的具体实现类，美国供电器——通过110V电压供电
class americanCharger implements Adaptee {
    @Override
    public void chargeBy110V() {
        System.out.println("美国供电器，只为你服务，正在通过110V电压为您充电");
    }
}

//抽象类适配器，通过继承现有接口来完成对现有接口的扩展
abstract class Adapter implements Target {
    Adaptee adaptee;//持有现有接口具体实现对象的引用

    public Adapter(Adaptee adaptee) {
        this.adaptee = adaptee;
    }
}

//中国自制
class ChinaMakeAdapter extends Adapter {

    public ChinaMakeAdapter(Adaptee adaptee) {
        super(adaptee);
    }

    @Override
    public void chargeBy220V() {
        adaptee.chargeBy110V();//该对象的现有功能
        System.out.println("再加110V，达到220V，认准中国制造，冲鸭！");//对现有功能扩展
    }
}

//测试类
```

```
public class Test {
    public static void main(String[] args) throws FileNotFoundException {
        //现在我们有一个美国110V供电站，但我们无法使用
        Adaptee adaptee = new americanCharger();

        //我们将这个供电站交给中国制造的适配器
        Adapter adapter = new ChinaMakeAdapter(adaptee);

        //接下来我们通过适配器充电就好了
        adapter.chargeBy220V();
    }
}
//输出同上
```

此外可以适配器还通过实现两个接口以达到双向适配的目的，即从接口A可以适配到接口B，从接口B也可以适配到接口A，这种情况并不常见。

```
//接口A——220V的电压供电
interface A {
    void chargeBy220V();
}


//接口A的具体实现类，中国供电器——通过220V电压供电
class ChinaCharger implements A {
    @Override
    public void chargeBy220V() {
        System.out.println("220V电压中国充电，值得信赖");
    }
}

//接口B——110V电压供电
interface B {
    void chargeBy110V();
}

//接口B的具体实现类，美国供电器——通过110V电压供电
class AmericanCharger implements B {
    @Override
    public void chargeBy110V() {
        System.out.println("美国充电器，只为你服务，正在通过110V电压为您充电");
    }
}

//双向适配器
class Adapter implements A, B {
    A a; //220V充电
    B b; //110V充电


    public Adapter(A a) {
        this.a = a;
    }
```

```
    public Adapter(B b) {
        this.b = b;
    }

    @Override
    public void chargeBy220V() {
        b.chargeBy110V(); //当前接口
        System.out.println("加码，加到220V！！");//适配目标接口
    }

    @Override
    public void chargeBy110V() {
        a.chargeBy220V();//当前接口
        System.out.println("缓冲电压，现在是110V了");
    }
}


//测试类
public class Test {
    public static void main(String[] args) throws FileNotFoundException {
        //我们去美国，酒店里有一个美国110V充电站，我们需要220V的电压
        B b = new AmericanCharger();
        //我们将这个充电站交给适配器以获取220V电压充电
        Adapter adapter1 = new Adapter(b);
        //接下来我们通过适配器充电就好了
        adapter1.chargeBy220V();

        System.out.println();

        //美国人来中国，酒店里有一个中国220V充电站，但他需要110V的电压
        A a = new ChinaCharger();
        //将这个充电站交给适配器以获取110V电压充电
        Adapter adapter2 = new Adapter(a);
        //接下来我们通过适配器充电就好了
        adapter2.chargeBy110V();
    }
}
//输出
/*
美国充电器，只为你服务，正在通过110V电压为您充电
加码，加到220V！！

220V电压中国充电，值得信赖
缓冲电压，现在是110V了

Process finished with exit code 0
*/
```

通过实现两个接口的方式，达到不同接口的双向适配，在某些情况下还是很实用的，例如TypeC—USB接口转换器，既能从typeC转USB也能从USB转typeC。

适配器模式总结

优点：

1. 可以让任何两个没有关联的类一起运行。
2. 提高了类的复用，可以一致化多个不同接口。
3. 将现有接口实现类隐藏，增加了类的透明度。
4. 灵活性高，可自由适配。

缺点：

1. 过多地使用适配器，会让系统非常零乱，不易整体进行把握。比如，明明看到调用的是 A 接口，其实内部被适配成了 B 接口的实现，一个系统如果太多出现这种情况，无异于一场灾难。因此如果不是很有必要，可以不使用适配器，而是直接对系统进行重构。
2. 某些适配工作可能非常困难，例如让房子飞起来。

当我们有动机地修改一个正常运行的系统的接口，这时应该考虑使用适配器模式。

注意事项：适配器不是在详细设计时添加的，而是解决正在服役的项目的问题，即现有接口可能无法改变(去美国不可能把人家110V电压供给改成220V电压供给)。