适配器模式

题目链接

适配器模式-扩展坞

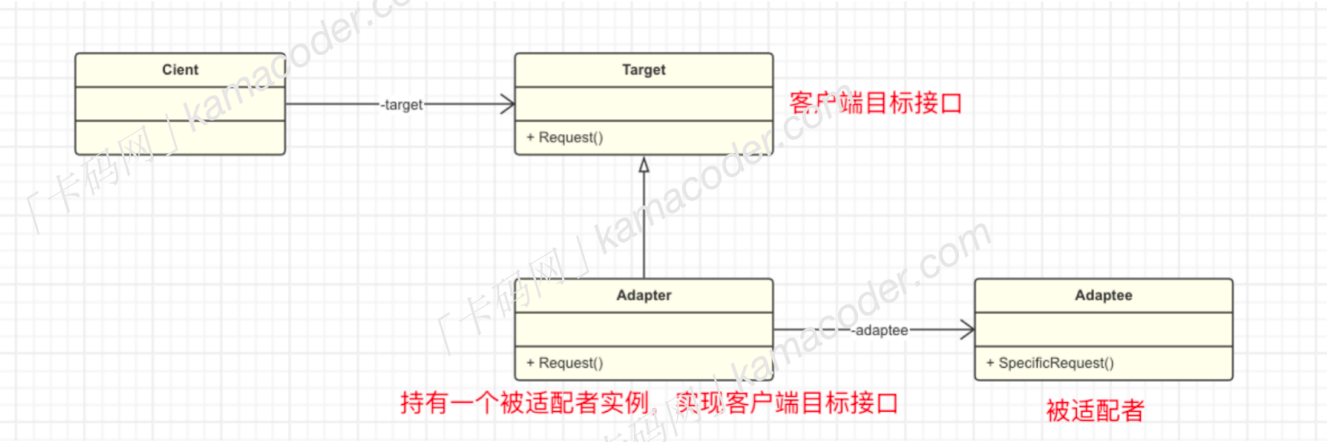
什么是适配器

适配器模式Adapter是一种结构型设计模式，它可以将一个类的接口转换成客户希望的另一个接口，主要目的是充当两个不同接口之间的桥梁，使得原本接口不兼容的类能够一起工作。

基本结构

适配器模式分为以下几个基本角色：

> *可以把适配器模式理解成拓展坞，起到转接的作用，原有的接口是USB，但是客户端需要使用type-c， 便使用拓展坞提供一个type-c接口给客户端使用*



- 目标接口Target：客户端希望使用的接口
- 适配器类Adapter：实现客户端使用的目标接口，持有一个需要适配的类实例。
- 被适配者Adaptee：需要被适配的类

这样，客户端就可以使用目标接口，而不需要对原来的Adaptee进行修改，Adapter起到一个转接扩展的作用。

基本实现

```
// 目标接口
interface Target {
    void request();
}

// 被适配者类
class Adaptee {
    void specificRequest() {
        System.out.println("Specific request");
    }
}

// 适配器类
class Adapter implements Target {
    // 持有一个被适配者实例
    private Adaptee adaptee;

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

    @Override
    public void request() {
        // 调用被适配者类的方法
        adaptee.specificRequest();
    }
}

// 客户端代码
public class Client {
    public static void main(String[] args) {
        Target target = new Adapter(new Adaptee());
        target.request();
    }
}
```

应用场景

在开发过程中，适配器模式往往扮演者“补救”和“扩展”的角色：

- 当使用一个已经存在的类，但是它的接口与你的代码不兼容时，可以使用适配器模式。
- 在系统扩展阶段需要增加新的类时，并且类的接口和系统现有的类不一致时，可以使用适配器模式。

使用适配器模式可以将客户端代码与具体的类解耦，客户端不需要知道被适配者的细节，客户端代码也不需要修改，这使得它具有良好的扩展性，但是这也势必导致系统变得更加复杂。

具体来说，适配器模式有着以下应用：

- 不同的项目和库可能使用不同的日志框架，不同的日志框架提供的API也不同，因此引入了适配器模式使得不同的API适配为统一接口。
- Spring MVC中，`HandlerAdapter` 接口是适配器模式的一种应用。它负责将处理器（Handler）适配到框架中，使得不同类型的处理器能够统一处理请求。
- 在.NET中，`DataAdapter` 用于在数据源（如数据库）和 `DataSet` 之间建立适配器，将数据从数据源适配到 `DataSet` 中，以便在.NET应用程序中使用。

本题代码

```
// 测试程序
import java.util.Scanner;
// USB 接口
interface USB {
    void charge();
}

// TypeC 接口
interface TypeC {
    void chargeWithTypeC();
}

// 适配器类
class TypeCAdapter implements USB {
    private TypeC typeC;

    public TypeCAdapter(TypeC typeC) {
        this.typeC = typeC;
    }

    @Override
    public void charge() {
        typeC.chargeWithTypeC();
    }
}

// 新电脑类，使用 TypeC 接口
class NewComputer implements TypeC {
    @Override
    public void chargeWithTypeC() {
        System.out.println("TypeC");
```

```
    }
}

// 适配器充电器类，使用 USB 接口
class AdapterCharger implements USB {
    @Override
    public void charge() {
        System.out.println("USB Adapter");
    }
}



public class Main {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);

        // 读取连接次数
        int N = scanner.nextInt();
        scanner.nextLine(); // 消耗换行符

        for (int i = 0; i < N; i++) {
            // 读取用户选择
            int choice = scanner.nextInt();

            // 根据用户的选择创建相应对象
            if (choice == 1) {
                TypeC newComputer = new NewComputer();
                newComputer.chargeWithTypeC();
            } else if (choice == 2) {
                USB usbAdapter = new AdapterCharger();
                usbAdapter.charge();
            }
        }
        scanner.close();
    }
}
```

其他语言代码

Java

类适配器模式代码：

```
import java.util.*;

// 定义计算机端口接口
interface ComputerPort {
    void connect();
}

// TypeC端口类实现ComputerPort接口
class TypeCPort implements ComputerPort {
    @Override
    public void connect() {
        System.out.println("TypeC");
    }
}

// USB设备类
class USBDevice {
    public void connectUSB() {
        System.out.println("USB Adapter");
    }
}

// TypeC到USB适配器类
class TypeCToUSBAdapter extends USBDevice implements ComputerPort {
    @Override
    public void connect() {
        super.connectUSB();
    }
}

public class Main{
    public static void main(String[] args) {
        Scanner inputScanner = new Scanner(System.in);

        Map<Integer, ComputerPort> connectionModes = new HashMap<>();
        connectionModes.put(1, new TypeCPort());
        connectionModes.put(2, new TypeCToUSBAdapter());

        int totalConnections = inputScanner.nextInt();
        inputScanner.nextLine();

        while (inputScanner.hasNextInt()) {
```

```
            int choice = inputScanner.nextInt();
            connectionModes.getOrDefault(choice, () ->
System.out.println("")).connect();
        }

        inputScanner.close();
    }
}
```

C++

```
#include <iostream>

// USB 接口
class USB {
public:
    virtual void charge() = 0;
};

// TypeC 接口
class TypeC {
public:
    virtual void chargeWithTypeC() = 0;
};

// 适配器类
class TypeCAdapter : public USB {
private:
    TypeC* typeC;

public:
    TypeCAdapter(TypeC* typeC) : typeC(typeC) {}

    void charge() override {
        typeC->chargeWithTypeC();
    }
};

// 新电脑类，使用 TypeC 接口
class NewComputer : public TypeC {
public:
    void chargeWithTypeC() override {
        std::cout << "TypeC" << std::endl;
    }
};
```

```cpp
// 适配器充电器类，使用 USB 接口
class AdapterCharger : public USB {
public:
    void charge() override {
        std::cout << "USB Adapter" << std::endl;
    }
};

int main() {
    // 读取连接次数
    int N;
    std::cin >> N;
    std::cin.ignore(); // 消耗换行符

    for (int i = 0; i < N; i++) {
        // 读取用户选择
        int choice;
        std::cin >> choice;

        // 根据用户的选择创建相应对象
        if (choice == 1) {
            TypeC* newComputer = new NewComputer();
            newComputer->chargeWithTypeC();
            delete newComputer;
        } else if (choice == 2) {
            USB* usbAdapter = new AdapterCharger();
            usbAdapter->charge();
            delete usbAdapter;
        }
    }

    return 0;
}
```

Python

```python
# USB 接口
class USB:
    def charge(self):
        pass

# TypeC 接口
class TypeC:
    def charge_with_type_c(self):
        pass
```

```python
# 适配器类
class TypeCAdapter(USB):
    def __init__(self, type_c):
        self.type_c = type_c

    def charge(self):
        self.type_c.charge_with_type_c()

# 新电脑类，使用 TypeC 接口
class NewComputer(TypeC):
    def charge_with_type_c(self):
        print("TypeC")

# 适配器充电器类，使用 USB 接口
class AdapterCharger(USB):
    def charge(self):
        print("USB Adapter")

if __name__ == "__main__":
    # 读取连接次数
    N = int(input())

    for _ in range(N):
        # 读取用户选择
        choice = int(input())

        # 根据用户的选择创建相应对象
        if choice == 1:
            new_computer = NewComputer()
            new_computer.charge_with_type_c()
        elif choice == 2:
            usb_adapter = AdapterCharger()
            usb_adapter.charge()
```

Go

```go
package main

import "fmt"

// USB 接口
type USB interface {
    charge()
}

// TypeC 接口
```

```go
type TypeC interface {
    chargeWithTypeC()
}

// 适配器类
type TypeCAdapter struct {
    typeC TypeC
}

func (tca *TypeCAdapter) charge() {
    tca.typeC.chargeWithTypeC()
}

// 新电脑类，使用 TypeC 接口
type NewComputer struct{}

func (nc *NewComputer) chargeWithTypeC() {
    fmt.Println("TypeC")
}

// 适配器充电器类，使用 USB 接口
type AdapterCharger struct{}

func (ac *AdapterCharger) charge() {
    fmt.Println("USB Adapter")
}

func main() {
    var N int
    fmt.Scan(&N) // 读取连接次数

    for i := 0; i < N; i++ {
        var choice int
        fmt.Scan(&choice) // 读取用户选择

        // 根据用户的选择创建相应对象
        if choice == 1 {
            newComputer := &NewComputer{}
            adapter := &TypeCAdapter{typeC: newComputer}
            adapter.charge()
        } else if choice == 2 {
            usbAdapter := &AdapterCharger{}
            usbAdapter.charge()
        }
    }
}
```