适配器模式

1. 引言

- **什么是设计模式**

  设计模式是软件设计中常见问题的典型解决方案。学习设计模式能提高代码的可复用性、可维护性、可扩展性。

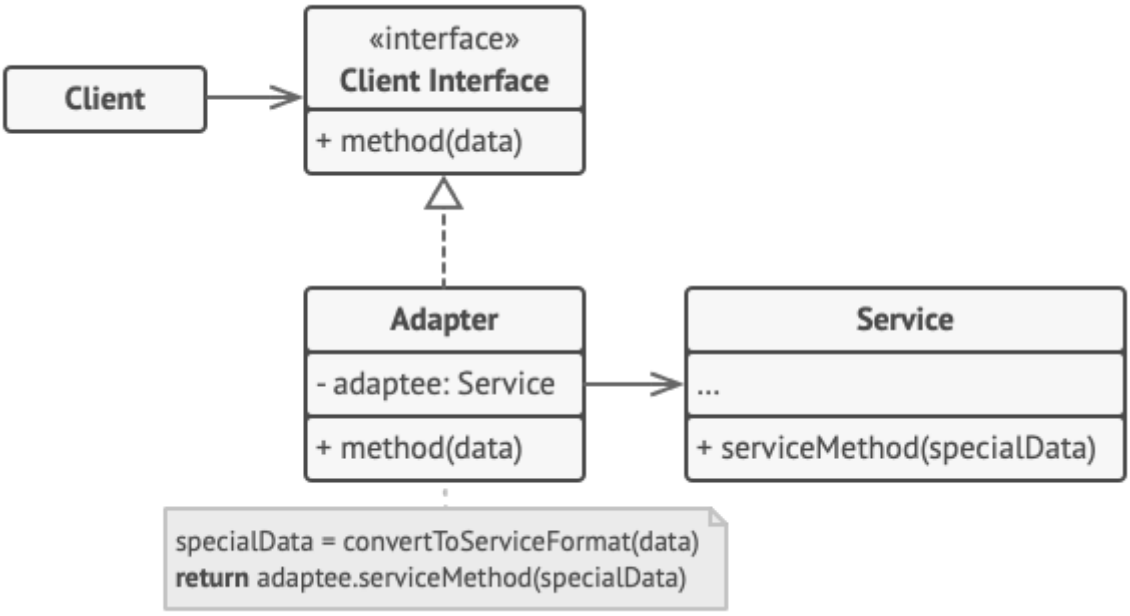
- **适配器模式**

  适配器模式的主要目的是为了将一个类的接口转换成客户期望的另一个接口。它使得原本由于接口不兼容而无法一起工作的类可以通过适配器实现互操作，增强了系统的灵活性和可重用性。

  生活中常见的电源适配器：

2. 适配器模式的组成

- **目标接口（Target）**：客户希望使用的接口。
- **适配者（Adapter）**：实现目标接口，并将请求委托给被适配者。
- **被适配者（Adaptee）**：存在的接口，但不兼容目标接口。



3.代码示例

```
/**
 * Target 定义客户使用的接口
 */

class Target {
 public:
  virtual ~Target() = default;

  virtual std::string Request() const {
    return "Target: the default target's behavior.";
  }
};

class Adaptee {
 public:
  std::string SpecificRequest() const {
    return ".eetpadA eht fo roivaheb laicepS";
  }
};

class Adapter : public Target {
 private:
  Adaptee *adaptee_;

 public:
  Adapter(Adaptee *adaptee) : adaptee_(adaptee) {}
  std::string Request() const override {
    std::string to_reverse = this->adaptee_->SpecificRequest();
    std::reverse(to_reverse.begin(), to_reverse.end());
    return "Adapter: (TRANSLATED) " + to_reverse;
  }
};

void ClientCode(const Target *target) {
  std::cout << target->Request();
}

int main() {
    std::cout << "Client: 调用 target 接口:\n";
    Target *target = new Target;
    ClientCode(target);
    std::cout << "\n\n";

    Adaptee *adaptee = new Adaptee;
    std::cout << "Client: 无法直接调用 Adaptee 的接口:\n";
    std::cout << "Adaptee: " << adaptee->SpecificRequest();
    std::cout << "\n\n";
    std::cout << "Client: 通过 Adapter 调用 Adaptee 的接口:\n";
    Adapter *adapter = new Adapter(adaptee);
    ClientCode(adapter);
    std::cout << "\n";

    delete target;
    delete adaptee;
```

```
    delete adapter;

    return 0;
}
```

执行结果

```
Client: 调用 target 接口:
Target: The default target's behavior.

Client: 无法直接调用 Adaptee 的接口:
Adaptee: .eetpadA eht fo roivaheb laicepS

Client: 通过 Adapter 调用 Adaptee 的接口:
Adapter: (TRANSLATED) Special behavior of the Adaptee.
```

4. 使用场景

- 封装有缺陷的接口设计

  假设我们依赖的外部系统在接口设计方面有缺陷，引入之后会影响到我们自身代码的可测试性。为了隔离设计上的缺陷，我们希望对外部系统提供的接口进行二次封装，抽象出更好的接口设计，这个时候就可以使用适配器模式了。

- 统一多个类的接口设计

  某个功能的实现依赖多个外部系统，通过适配器模式，将它们的接口适配为统一的接口定义，然后我们就可以使用多态的特性来复用代码逻辑。

- 替换依赖的外部系统
  当我们把项目中依赖的一个外部系统替换为另一个外部系统的时候，利用适配器模式，可以减少对代码的改动。具体的代码示例如下所示：

- 兼容老版本接口

  适配器模式允许我们创建一个中间层类， 其可作为代码与遗留类、 第三方类之间的转换器。

5. 优缺点

- 优点
  - 单一职责原则

    可以将接口或数据转换代码从程序主要业务逻辑中分离。

  - 开闭原则

    只要客户端代码通过客户端接口与适配器进行交互， 就能在不修改现有客户端代码的情况下在程序中添加新类型的适配器。

- 缺点
  - 代码整体复杂度增加，会导致难以维护

    因为需要新增一系列接口和类。 有时直接更改服务类使其与其他代码兼容会更简单。