代理模式

题目链接

代理模式-小明买房子

基本概念

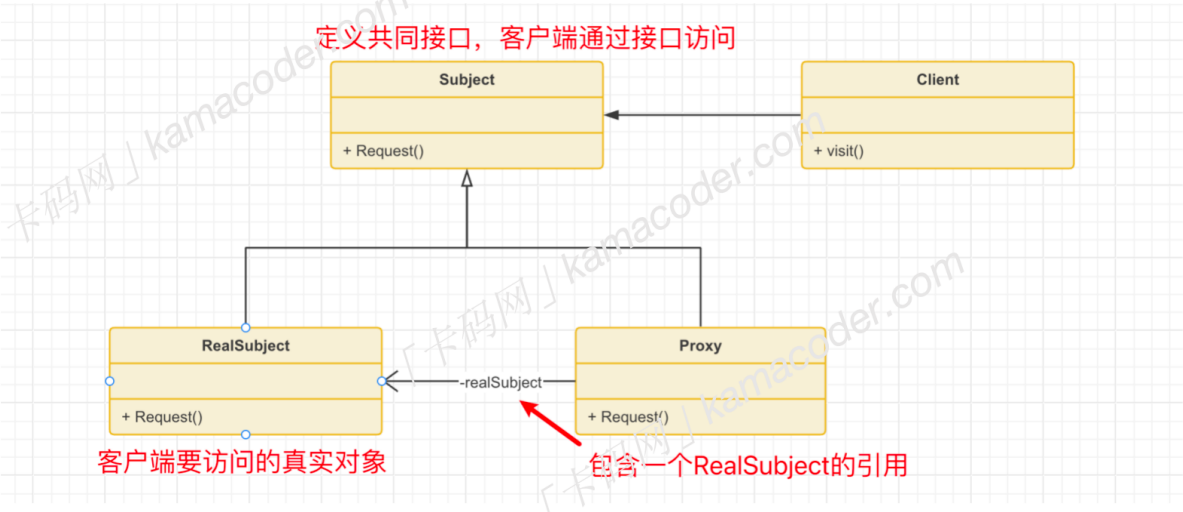
代理模式Proxy Pattern是一种结构型设计模式，用于控制对其他对象的访问。

在代理模式中，允许一个对象（代理）充当另一个对象（真实对象）的接口，以控制对这个对象的访问。通常用于在访问某个对象时引入一些间接层(中介的作用)，这样可以在访问对象时添加额外的控制逻辑，比如限制访问权限，延迟加载。

比如说有一个文件加载的场景，为了避免直接访问“文件”对象，我们可以新增一个代理对象，代理对象中有一个对“文件对象”的引用，在代理对象的 `load` 方法中，可以在访问真实的文件对象之前进行一些操作，比如权限检查，然后调用真实文件对象的 `load` 方法，最后在访问真实对象后进行其他操作，比如记录访问日志。

基本结构

代理模式的主要角色有:



- Subject（抽象主题）： 抽象类，通过接口或抽象类声明真实主题和代理对象实现的业务方法。
- RealSubject（真实主题）：定义了Proxy所代表的真实对象，是客户端最终要访问的对象。

- Proxy（代理）：包含一个引用，该引用可以是RealSubject的实例，控制对RealSubject的访问，并可能负责创建和删除RealSubject的实例。

实现方式

代理模式的基本实现分为以下几个步骤：

1. 定义抽象主题，一般是接口或者抽象类，声明真实主题和代理对象实现的业务方法。

```
// 1. 定义抽象主题
interface Subject {
    void request();
}
```

2. 定义真实主题，实现抽象主题中的具体业务

```
// 2. 定义真实主题
class RealSubject implements Subject {
    @Override
    public void request() {
        System.out.println("RealSubject handles the request.");
    }
}
```

3. 定义代理类，包含对RealSubject的引用，并提供和真实主题相同的接口，这样代理就可以替代真实主题，并对真实主题进行功能扩展。

```
// 3. 定义代理
class Proxy implements Subject {
    // 包含一个引用
    private RealSubject realSubject;

    @Override
    public void request() {
        // 在访问真实主题之前可以添加额外的逻辑
        if (realSubject == null) {
            realSubject = new RealSubject();
        }
        // 调用真实主题的方法
        realSubject.request();

        // 在访问真实主题之后可以添加额外的逻辑
    }
}
```

4. 客户端使用代理

```
// 4. 客户端使用代理
public class Main {
    public static void main(String[] args) {
        // 使用代理
        Subject proxy = new Proxy();
        proxy.request();
    }
}
```

使用场景

代理模式可以控制客户端对真实对象的访问，从而限制某些客户端的访问权限，此外代理模式还常用在访问真实对象之前或之后执行一些额外的操作（比如记录日志），对功能进行扩展。

以上特性决定了代理模式在以下几个场景中有着广泛的应用：

- 虚拟代理：当一个对象的创建和初始化比较昂贵时，可以使用虚拟代理，虚拟代理可以延迟对象的实际创建和初始化，只有在需要时才真正创建并初始化对象。
- 安全代理：安全代理可以根据访问者的权限决定是否允许访问真实对象的方法。

但是代理模式涉及到多个对象之间的交互，引入代理模式会增加系统的复杂性，在需要频繁访问真实对象时，还可能会有一些性能问题。

代理模式在许多工具和库中也有应用：

- Spring 框架的 AOP 模块使用了代理模式来实现切面编程。通过代理，Spring 能够在目标对象的方法执行前、执行后或抛出异常时插入切面逻辑，而不需要修改原始代码。
- Java 提供了动态代理机制，允许在运行时生成代理类。
- Android中的Glide框架使用了代理模式来实现图片的延迟加载。

本题代码

```
import java.util.Scanner;

// 抽象主题
interface HomePurchase {
    void requestPurchase(int area);
}

//真实主题
class HomeBuyer implements HomePurchase {
    @Override
    public void requestPurchase(int area) {
```

```
            System.out.println("YES");
        }
    }

    // 代理类
    class HomeAgentProxy implements HomePurchase {
        private HomeBuyer homeBuyer = new HomeBuyer();

        @Override
        public void requestPurchase(int area) {
            if (area > 100) {
                homeBuyer.requestPurchase(area);
            } else {
                System.out.println("NO");
            }
        }
    }

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

            int n = scanner.nextInt();
            for (int i = 0; i < n; i++) {
                int area = scanner.nextInt();
                buyerProxy.requestPurchase(area);
            }

            scanner.close();
        }
    }
```

扩展：代理模式和适配器模式有什么区别

代理模式的主要目的是控制对对象的访问。通常用于在访问真实对象时引入一些额外的控制逻辑，如权限控制、延迟加载等。

适配器模式的主要目的是使接口不兼容的对象能够协同工作。适配器模式允许将一个类的接口转换成另一个类的接口，使得不同接口的类可以协同工作。

其他语言代码

C++

```cpp
#include <iostream>

// 抽象主题
class HomePurchase {
public:
    virtual void requestPurchase(int area) = 0;
};

// 真实主题
class HomeBuyer : public HomePurchase {
public:
    void requestPurchase(int area) override {
        std::cout << "YES" << std::endl;
    }
};

// 代理类
class HomeAgentProxy : public HomePurchase {
private:
    HomeBuyer homeBuyer;

public:
    void requestPurchase(int area) override {
        if (area > 100) {
            homeBuyer.requestPurchase(area);
        } else {
            std::cout << "NO" << std::endl;
        }
    }
};

int main() {
    HomePurchase* buyerProxy = new HomeAgentProxy();

    int n;
    std::cin >> n;

    for (int i = 0; i < n; i++) {
        int area;
        std::cin >> area;
        buyerProxy->requestPurchase(area);
```

```
    }

    delete buyerProxy;

    return 0;
}
```

Python

```python
# 抽象主题
class HomePurchase:
    def request_purchase(self, area):
        pass

# 真实主题
class HomeBuyer(HomePurchase):
    def request_purchase(self, area):
        print("YES")

# 代理类
class HomeAgentProxy(HomePurchase):
    def __init__(self):
        self.home_buyer = HomeBuyer()

    def request_purchase(self, area):
        if area > 100:
            self.home_buyer.request_purchase(area)
        else:
            print("NO")

if __name__ == "__main__":
    buyer_proxy = HomeAgentProxy()

    n = int(input())
    for _ in range(n):
        area = int(input())
        buyer_proxy.request_purchase(area)
```

Go

```go
package main

import "fmt"

// 抽象主题
```

```go
type HomePurchase interface {
    requestPurchase(area int)
}

// 真实主题
type HomeBuyer struct{}

func (hb *HomeBuyer) requestPurchase(area int) {
    fmt.Println("YES")
}

// 代理类
type HomeAgentProxy struct {
    homeBuyer HomePurchase
}

func (hap *HomeAgentProxy) requestPurchase(area int) {
    if area > 100 {
        hap.homeBuyer.requestPurchase(area)
    } else {
        fmt.Println("NO")
    }
}

func main() {
    var buyerProxy HomePurchase = &HomeAgentProxy{homeBuyer: &HomeBuyer{}}

    var n int
    fmt.Scan(&n)

    for i := 0; i < n; i++ {
        var area int
        fmt.Scan(&area)
        buyerProxy.requestPurchase(area)
    }
}
```