中介者模式

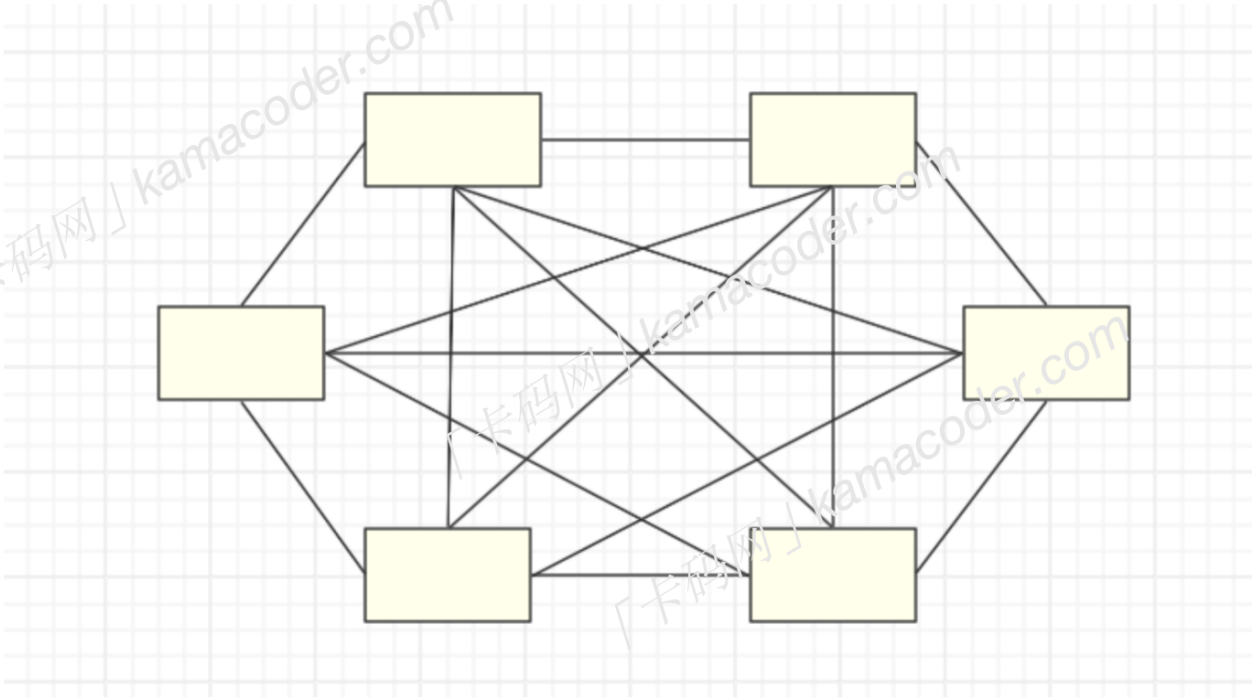
题目链接

中介者模式-简易聊天室

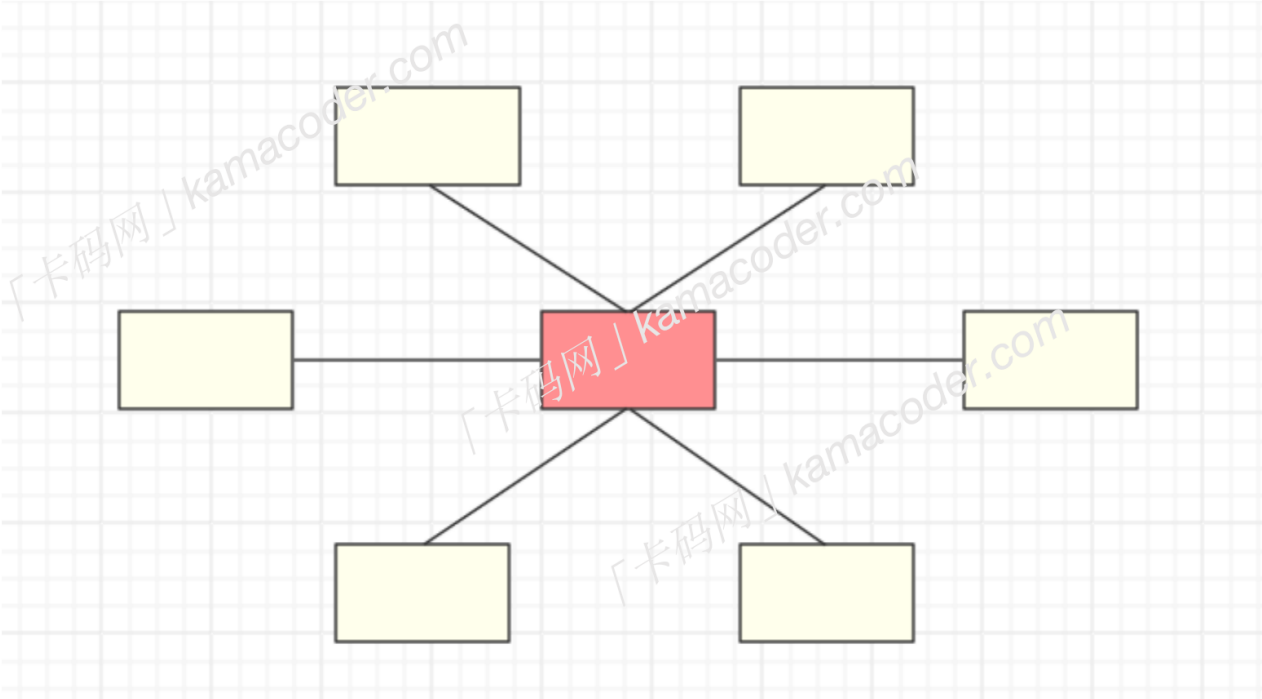
基本概念

中介者模式（Mediator Pattern）也被称为调停者模式，是一种行为型设计模式，它通过一个中介对象来封装一组对象之间的交互，从而使这些对象不需要直接相互引用。这样可以降低对象之间的耦合度，使系统更容易维护和扩展。

当一个系统中的对象有很多且多个对象之间有复杂的相互依赖关系时，其结构图可能是下面这样的。

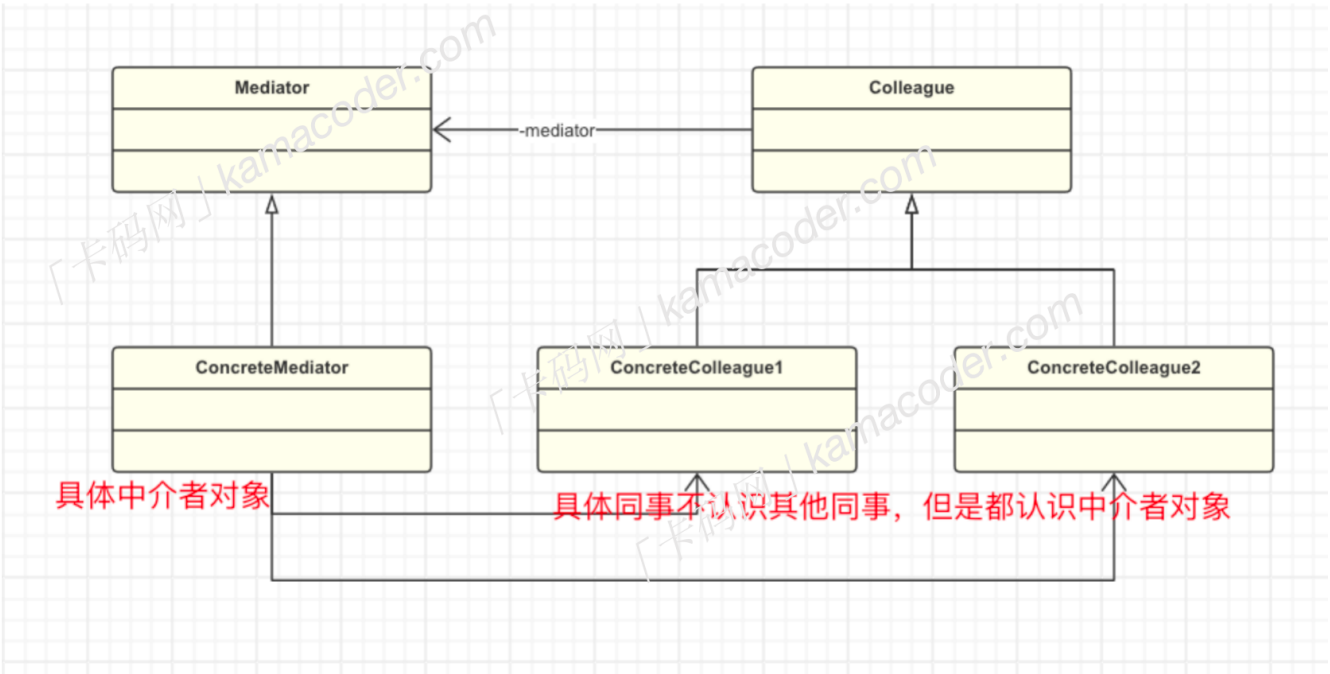


这种依赖关系很难理清，这时我们可以引入一个中介者对象来进行协调和交互。中介者模式可以使得系统的网状结构变成以中介者为中心的星形结构，每个具体对象不再通过直接的联系与另一个对象发生相互作用，而是通过“中介者”对象与另一个对象发生相互作用。



基本结构

中介者模式包括以下几个重要角色：



- **抽象中介者（Mediator）：** 定义中介者的接口，用于各个具体同事对象之间的通信。

- **具体中介者（Concrete Mediator）：** 实现抽象中介者接口，负责协调各个具体同事对象的交互关系，它需要知道所有具体同事类，并从具体同事接收消息，向具体同事对象发出命令。
- **抽象同事类（Colleague）：** 定义同事类的接口，维护一个对中介者对象的引用，用于通信。
- **具体同事类（Concrete Colleague）：** 实现抽象同事类接口，每个具体同事类只知道自己的行为，而不了解其他同事类的情况，因为它们都需要与中介者通信，通过中介者协调与其他同事对象的交互。

简易实现

```
// 抽象中介者
public abstract class Mediator {
    void register(Colleague colleague);
    // 定义一个抽象的发送消息方法
    public abstract void send(String message, Player player);

}

// 具体中介者
public class ConcreteMediator extends Mediator {
    private List<Colleague> colleagues = new ArrayList<>();

    public void register((Colleague colleague) {
        colleagues.add(colleague);
    }

    @Override
    public void send(String message, Colleague colleague) {
        for (Colleague c : colleagues) {
            // 排除发送消息的同事对象
            if (c != colleague) {
                c.receive(message);
            }
        }
    }
}

// 同事对象
abstract class Colleague {
    protected Mediator mediator;

    public Colleague(Mediator mediator) {
        this.mediator = mediator;
```

```
    }

    // 发送消息
    public abstract void send(String message);

    // 接收消息
    public abstract void receive(String message);
}

// 具体同事对象1
class ConcreteColleague1 extends Colleague {
    public ConcreteColleague1(Mediator mediator) {
        super(mediator);
    }

    @Override
    public void send(String message) {
        mediator.send(message, this);
    }

    @Override
    public void receive(String message) {
        System.out.println("ConcreteColleague1 received: " + message);
    }
}

// 具体同事对象2
class ConcreteColleague2 extends Colleague {
    public ConcreteColleague2(Mediator mediator) {
        super(mediator);
    }

    @Override
    public void send(String message) {
        mediator.send(message, this);
    }

    @Override
    public void receive(String message) {
        System.out.println("ConcreteColleague2 received: " + message);
    }
}

// 客户端
public class Main{
    public static void main(String[] args) {
```

```
        // 创建中介者
        Mediator mediator = new ConcreteMediator();

        // 创建同事对象
        Colleague colleague1 = new ConcreteColleague1(mediator);
        Colleague colleague2 = new ConcreteColleague2(mediator);

        // 注册同事对象到中介者
        mediator.register(colleague1);
        mediator.register(colleague2);

        // 同事对象之间发送消息
        colleague1.send("Hello from Colleague1!");
        colleague2.send("Hi from Colleague2!");
    }
}
```

使用场景

中介者模式使得同事对象不需要知道彼此的细节，只需要与中介者进行通信，简化了系统的复杂度，也降低了各对象之间的耦合度，但是这也会使得中介者对象变得过于庞大和复杂，如果中介者对象出现问题，整个系统可能会受到影响。

中介者模式适用于当系统对象之间存在复杂的交互关系或者系统需要在不同对象之间进行灵活的通信时使用，可以使得问题简化，

本题代码

```
import java.util.*;

// 抽象中介者
interface ChatRoomMediator {
    void sendMessage(String sender, String message);
    void addUser(ChatUser user);
    Map<String, ChatUser> getUsers();
}

// 具体中介者
class ChatRoomMediatorImpl implements ChatRoomMediator {
    private Map<String, ChatUser> users = new LinkedHashMap<>();

    @Override
    public void sendMessage(String sender, String message) {
        for (ChatUser user : users.values()) {
            if (!user.getName().equals(sender)) {
                user.receiveMessage(sender, message);
```

```
            }
        }
    }

    @Override
    public void addUser(ChatUser user) {
        users.put(user.getName(), user);
    }

    @Override
    public Map<String, ChatUser> getUsers() {
        return users;
    }
}

// 抽象同事类
abstract class ChatUser {
    private String name;
    private ChatRoomMediator mediator;
    private List<String> receivedMessages = new ArrayList<>();

    public ChatUser(String name, ChatRoomMediator mediator) {
        this.name = name;
        this.mediator = mediator;
        mediator.addUser(this);
    }

    public String getName() {
        return name;
    }

    public void sendMessage(String message) {
        mediator.sendMessage(name, message);
    }

    public abstract void receiveMessage(String sender, String message);

    public List<String> getReceivedMessages() {
        return receivedMessages;
    }

    protected void addReceivedMessage(String message) {
        receivedMessages.add(message);
    }
}
```

```java
// 具体同事类
class ConcreteChatUser extends ChatUser {
    public ConcreteChatUser(String name, ChatRoomMediator mediator) {
        super(name, mediator);
    }

    @Override
    public void receiveMessage(String sender, String message) {
        String receivedMessage = getName() + " received: " + message;
        addReceivedMessage(receivedMessage);
        System.out.println(receivedMessage);
    }
}

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

        int N = scanner.nextInt();
        List<String> userNames = new ArrayList<>();
        for (int i = 0; i < N; i++) {
            userNames.add(scanner.next());
        }

        ChatRoomMediator mediator = new ChatRoomMediatorImpl();

        // 创建用户对象
        for (String userName : userNames) {
            new ConcreteChatUser(userName, mediator);
        }

        // 发送消息并输出
        while (scanner.hasNext()) {
            String sender = scanner.next();
            String message = scanner.next();

            ChatUser user = mediator.getUsers().get(sender);
            if (user != null) {
                user.sendMessage(message);
            }
        }

        scanner.close();
    }
}
```

扩展：和代理模式的区别

中介者模式（Mediator Pattern）和代理模式（Proxy Pattern）在某些表述上有些类似，但是他们是完全不同的两个设计模式，中介者模式的目的是降低系统中各个对象之间的直接耦合，通过引入一个中介者对象，使对象之间的通信集中在中介者上。而在代理模式中，客户端通过代理与目标对象进行通信。代理可以在调用目标对象的方法前后进行一些额外的操作，其目的是控制对对象的访问，它们分别解决了不同类型的问题。

其他语言版本

C++

```
#include <iostream>
#include <vector>
#include <map>
#include <list>

// 抽象中介者
class ChatRoomMediator;

// 抽象同事类
class ChatUser {
private:
    std::string name;
    ChatRoomMediator* mediator;
    std::list<std::string> receivedMessages;

public:
    ChatUser(const std::string& name, ChatRoomMediator* mediator);

    std::string getName() const {
        return name;
    }

    void sendMessage(const std::string& message);

    virtual void receiveMessage(const std::string& sender, const
std::string& message) = 0;

    std::list<std::string> getReceivedMessages() const {
        return receivedMessages;
    }
```

```
protected:
    void addReceivedMessage(const std::string& message) {
        receivedMessages.push_back(message);
    }
};

// 具体同事类
class ConcreteChatUser : public ChatUser {
public:
    ConcreteChatUser(const std::string& name, ChatRoomMediator* mediator);

    void receiveMessage(const std::string& sender, const std::string&
message) override;
};

// 抽象中介者
class ChatRoomMediator {
public:
    virtual void sendMessage(const std::string& sender, const std::string&
message) = 0;
    virtual void addUser(ChatUser* user) = 0;
    virtual std::map<std::string, ChatUser*> getUsers() = 0;
    virtual ~ChatRoomMediator() = default;
};

// 具体中介者
class ChatRoomMediatorImpl : public ChatRoomMediator {
private:
    std::map<std::string, ChatUser*> users;

public:
    void sendMessage(const std::string& sender, const std::string& message)
override {
        for (const auto& userPair : users) {
            if (userPair.first != sender) {
                userPair.second->receiveMessage(sender, message);
            }
        }
    }

    void addUser(ChatUser* user) override {
        users[user->getName()] = user;
    }

    std::map<std::string, ChatUser*> getUsers() override {
```

```
        return users;
    }
};

// 实现 ChatUser 类的成员函数
ChatUser::ChatUser(const std::string& name, ChatRoomMediator* mediator) :
name(name), mediator(mediator) {
    mediator->addUser(this);
}

void ChatUser::sendMessage(const std::string& message) {
    mediator->sendMessage(name, message);
}

// 实现 ConcreteChatUser 类的成员函数
ConcreteChatUser::ConcreteChatUser(const std::string& name,
ChatRoomMediator* mediator) : ChatUser(name, mediator) {}

void ConcreteChatUser::receiveMessage(const std::string& sender, const
std::string& message) {
    std::string receivedMessage = getName() + " received: " + message;
    addReceivedMessage(receivedMessage);
    std::cout << receivedMessage << std::endl;
}

int main() {
    std::vector<std::string> userNames;
    int N;
    std::cin >> N;

    for (int i = 0; i < N; i++) {
        std::string userName;
        std::cin >> userName;
        userNames.push_back(userName);
    }

    ChatRoomMediator* mediator = new ChatRoomMediatorImpl();

    // 创建用户对象
    for (const auto& userName : userNames) {
        new ConcreteChatUser(userName, mediator);
    }

    // 发送消息并输出
    std::string sender, message;
    while (std::cin >> sender >> message) {
```

```
        ChatUser* user = mediator->getUsers()[sender];
        if (user != nullptr) {
            user->sendMessage(message);
        }
    }

    delete mediator; // 释放中介者对象

    return 0;
}
```

Python

```
class ChatRoomMediator:
    def __init__(self):
        self.users = {}

    def send_message(self, sender, message):
        for user in self.users.values():
            if user.name != sender:
                user.receive_message(sender, message)

    def add_user(self, user):
        self.users[user.name] = user

class ChatUser:
    def __init__(self, name, mediator):
        self.name = name
        self.mediator = mediator
        self.received_messages = []

    def send_message(self, message):
        self.mediator.send_message(self.name, message)

    def receive_message(self, sender, message):
        received_message = f"{self.name} received: {message}"
        self.received_messages.append(received_message)
        print(received_message)

if __name__ == "__main__":
    # 读取用户数量
    N = int(input())

    # 读取用户列表
    user_names = input().split()
```

```
    # 创建中介者
    mediator = ChatRoomMediator()

    # 创建用户对象并注册到中介者
    for user_name in user_names:
        user = ChatUser(user_name, mediator)
        mediator.add_user(user)

    # 处理消息输入
    while True:
        try:
            sender, message = input().split()
            user = mediator.users.get(sender)
            if user:
                user.send_message(message)

        except EOFError:
            break
```

Go

```
package main

import "fmt"

type ChatRoomMediator struct {
    users map[string]*ChatUser
}

func NewChatRoomMediator() *ChatRoomMediator {
    return &ChatRoomMediator{
        users: make(map[string]*ChatUser),
    }
}

func (m *ChatRoomMediator) SendMessage(sender, message string) {
    for _, user := range m.users {
        if user.name != sender {
            user.ReceiveMessage(sender, message)
        }
    }
}

func (m *ChatRoomMediator) AddUser(user *ChatUser) {
    m.users[user.name] = user
}
```

```go
type ChatUser struct {
    name             string
    mediator         *ChatRoomMediator
    receivedMessages []string
}

func NewChatUser(name string, mediator *ChatRoomMediator) *ChatUser {
    return &ChatUser{
        name:             name,
        mediator:         mediator,
        receivedMessages: make([]string, 0),
    }
}

func (u *ChatUser) SendMessage(message string) {
    u.mediator.SendMessage(u.name, message)
}

func (u *ChatUser) ReceiveMessage(sender, message string) {
    receivedMessage := fmt.Sprintf("%s received: %s", u.name, message)
    u.receivedMessages = append(u.receivedMessages, receivedMessage)
    fmt.Println(receivedMessage)
}

func main() {
    var N int
    fmt.Scan(&N)

    userNames := make([]string, N)
    for i := 0; i < N; i++ {
        fmt.Scan(&userNames[i])
    }

    mediator := NewChatRoomMediator()

    for _, userName := range userNames {
        user := NewChatUser(userName, mediator)
        mediator.AddUser(user)
    }

    for {
        var sender, message string
        _, err := fmt.Scan(&sender, &message)
        if err != nil {
            break
```

```
        }

        user, ok := mediator.users[sender]
        if ok {
            user.SendMessage(message)
        }
    }
}
```