迭代器模式 备忘录模式

中介者模式

中介者模式（Mediator Pattern）是用来降低多个对象和类之间的通信复杂性。这种模式提供了一个中介类，该类通常处理不同类之间的通信，并支持松耦合，使代码易于维护。中介者模式属于行为型模式。

介绍

意图：用一个中介对象来封装一系列的对象交互，中介者使各对象不需要显式地相互引用，从而使其耦合松散，而且可以独立地改变它们之间的交互。

主要解决：对象与对象之间存在大量的关联关系，这样势必会导致系统的结构变得很复杂，同时若一个对象发生改变，我们也需要跟踪与之相关联的对象，同时做出相应的处理。

何时使用：多个类相互耦合，形成了网状结构。

如何解决：将上述网状结构分离为星型结构。

关键代码：对象 Colleague 之间的通信封装到一个类中单独处理。

应用实例： 1、中国加入 WTO 之前是各个国家相互贸易，结构复杂，现在是各个国家通过 WTO 来互相贸易。 2、机场调度系统。 3、MVC 框架，其中C（控制器）就是 M（模型）和 V（视图）的中介者。

优点： 1、降低了类的复杂度，将一对多转化成了一对一。 2、各个类之间的解耦。 3、符合迪米特原则。

缺点：中介者会庞大，变得复杂难以维护。

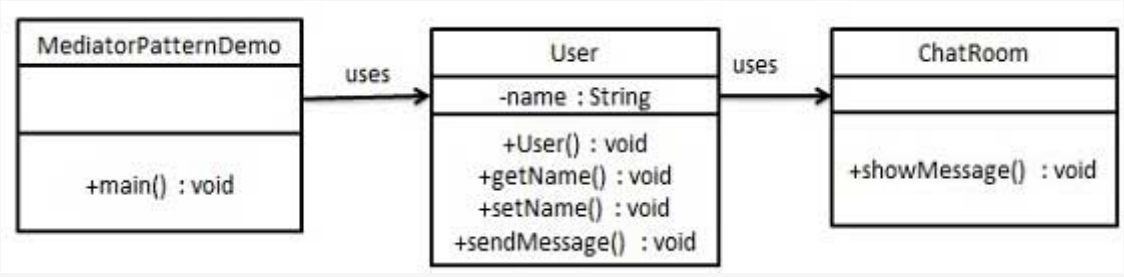
使用场景： 1、系统中对象之间存在比较复杂的引用关系，导致它们之间的依赖关系结构混乱而且难以复用该对象。 2、想通过一个中间类来封装多个类中的行为，而又不想生成太多的子类。

注意事项：不应当在职责混乱的时候使用。

实现

我们通过聊天室实例来演示中介者模式。实例中，多个用户可以向聊天室发送消息，聊天室向所有的用户显示消息。我们将创建两个类 *ChatRoom* 和 *User*。*User* 对象使用 *ChatRoom* 方法来分享他们的消息。

MediatorPatternDemo，我们的演示类使用 *User* 对象来显示他们之间的通信。



步骤 1

创建中介类。

ChatRoom.java

```
import java.util.Date;
public class ChatRoom {
public static void showMessage(User user, String message){
```

```
System.out.println(new Date().toString()
+ " [" + user.getName() +"] : " + message);
}
}
```

步骤 2

创建 user 类。

User.java

```
public class User {
private String name;
public String getName() {
return name;
}
public void setName(String name) {
this.name = name;
}
public User(String name){
this.name = name;
}
public void sendMessage(String message){
ChatRoom.showMessage(this,message);
}
}
```

步骤 3

使用 *User* 对象来显示他们之间的通信。

MediatorPatternDemo.java

```
public class MediatorPatternDemo {
public static void main(String[] args) {
User robert = new User("Robert");
User john = new User("John");
robert.sendMessage("Hi! John!");
john.sendMessage("Hello! Robert!");
}
}
```

步骤 4

执行程序，输出结果：

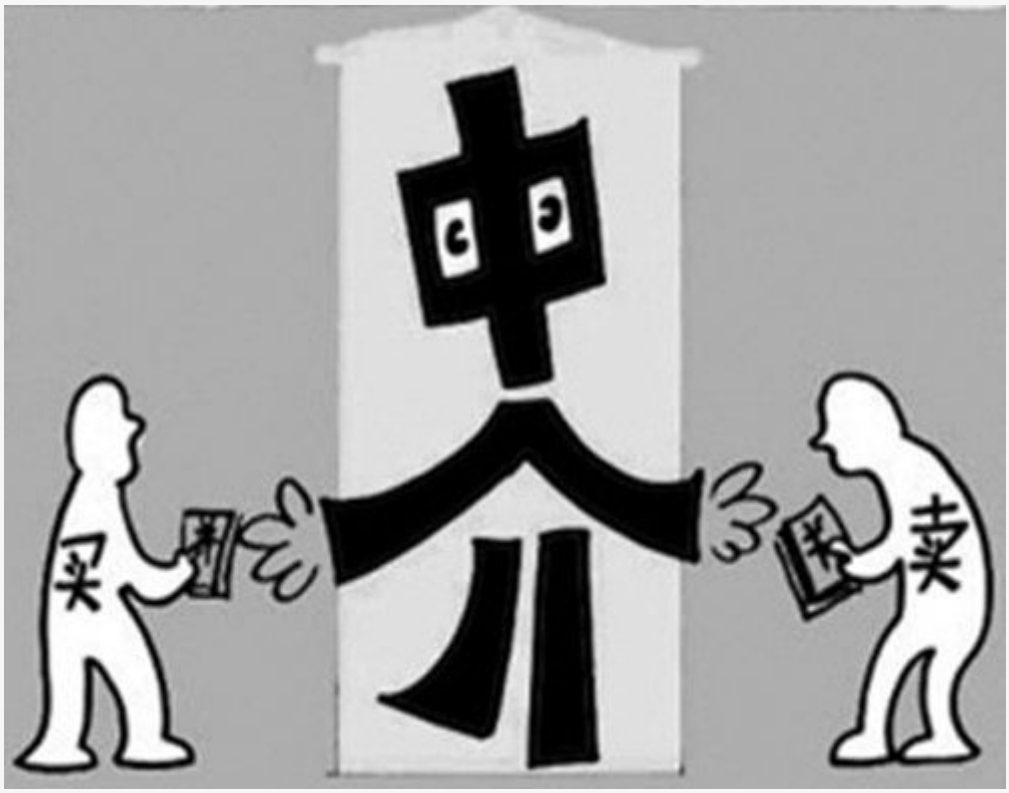
```
Thu Jan 31 16:05:46 IST 2013 [Robert] : Hi! John!
Thu Jan 31 16:05:46 IST 2013 [John] : Hello! Robert!
```

← 迭代器模式 备忘录模式 →

1 篇笔记

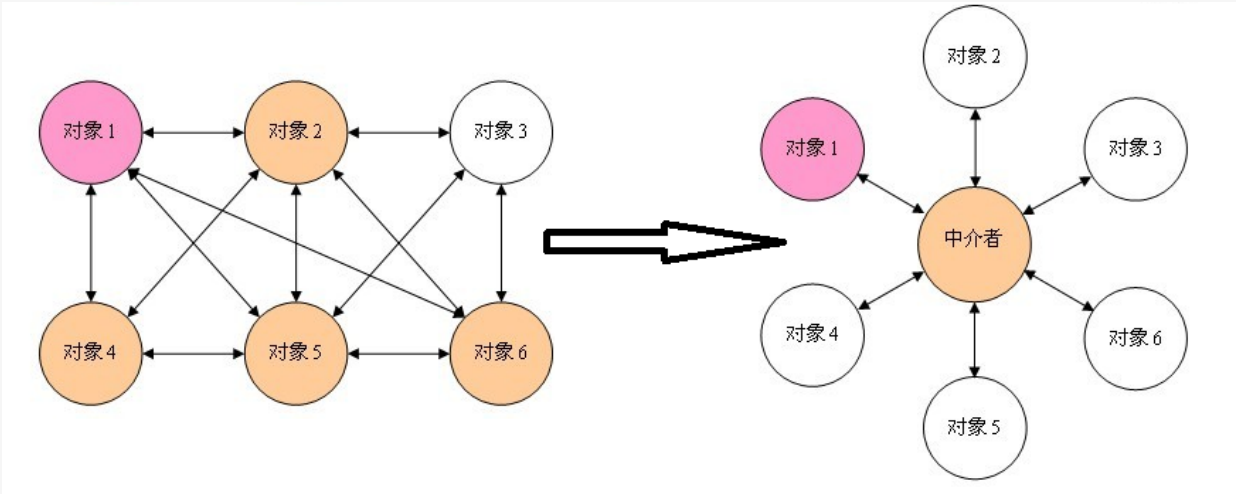
写笔记

什么是中介者模式？

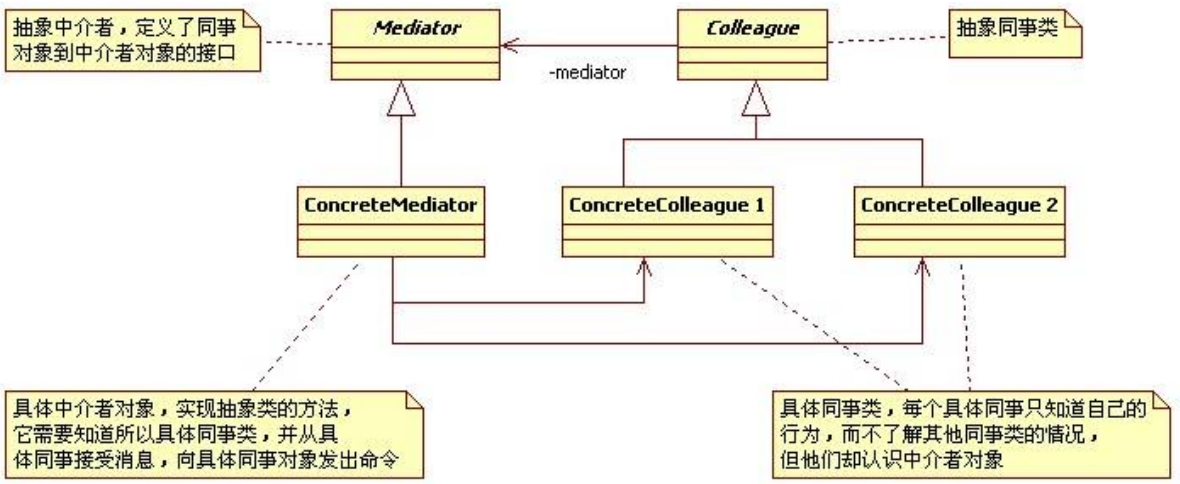


在现实生活中，有很多中介者模式的身影，例如QQ游戏平台，聊天室、QQ群、短信平台和房产中介。不论是QQ游戏还是QQ群，它们都是充当一个中间平台，QQ用户可以登录这个中间平台与其他QQ用户进行交流，如果没有这些中间平台，我们如果想与朋友进行聊天的话，可能就需要当面才可以了。电话、短信也同样是一个中间平台，有了这个中间平台，每个用户都不要直接依赖与其他用户，只需要依赖这个中间平台就可以了，一切操作都由中间平台去分发。

中介者模式，定义了一个中介对象来封装一系列对象之间的交互关系。中介者使各个对象之间不需要显式地相互引用，从而使耦合性降低，而且可以独立地改变它们之间的交互行为。



设计思路及代码实现:



以现实生活中打牌的例子来实现下中介者模式。打牌总有输赢，对应的则是货币的变化，如果不用中介者模式的话，实现如下：

```
/// <summary>
/// 抽象牌友类
/// </summary>
public abstract class AbstractCardPartner
{
    public int Money { get; set; }

    public abstract void ChangeMoney(int money, AbstractCardPartner other);
}

/// <summary>
/// 牌友A
/// </summary>
public class PartnerA : AbstractCardPartner
{
    public override void ChangeMoney(int money, AbstractCardPartner other)
    {
        Money += money;
        other.Money -= money;
    }
}

/// <summary>
/// 牌友B
/// </summary>
public class PartnerB : AbstractCardPartner
{
    public override void ChangeMoney(int money, AbstractCardPartner other)
    {
        Money += money;
        other.Money -= money;
    }
```

```
}

/// <summary>
/// 调用
/// </summary>
/// <param name="args"></param>
static void Main(string[] args)
{
    AbstractCardPartner A = new PartnerA();
    A.Money = 20;
    AbstractCardPartner B = new PartnerB();
    B.Money = 20;

    // A赢了B的钱减少
    A.ChangeMoney(5, B);
    Console.WriteLine("A 现在的钱是：{0}", A.Money); // 应该是25
    Console.WriteLine("B 现在的钱是：{0}", B.Money); // 应该是15

    // B赢了A的钱减少
    B.ChangeMoney(10, A);
    Console.WriteLine("A 现在的钱是：{0}", A.Money); // 应该是15
    Console.WriteLine("B 现在的钱是：{0}", B.Money); // 应该是25

    Console.ReadLine();
}
```

这样的实现确实解决了上面场景中的问题，并且使用了抽象类使具体牌友A和牌友B都依赖于抽象类，从而降低了同事类之间的耦合度。但是如果其中牌友A发生变化时，此时就会影响到牌友B的状态，如果涉及的对象变多的话，这时候某一个牌友的变化将会影响到其他所有相关联的牌友状态。例如牌友A算错了钱，这时候牌友A和牌友B的钱数都不正确了，如果是多个人打牌的话，影响的对象就会更多。这时候就会思考——能不能把算钱的任务交给程序或者算数好的人去计算呢，这时候就有了我们QQ游戏中的欢乐斗地主等牌类游戏了。

进一步完善的方案，即加入一个中介者对象来协调各个对象之间的关联，这也就是中介者模式的应用了，具体完善后的实现代码如下所示：

```
/// <summary>
/// 抽象牌友类
/// </summary>
public abstract class AbstractCardPartner
{
    public int Money { get; set; }

    public abstract void ChangeMoney(int money, AbstractMediator mediator);
}

/// <summary>
/// 牌友A
/// </summary>
public class PartnerA : AbstractCardPartner
```

```
{
    public override void ChangeMoney(int money, AbstractMediator mediator)
    {
        mediator.AWin(money);
    }
}

/// <summary>
/// 牌友B
/// </summary>
public class PartnerB : AbstractCardPartner
{
    public override void ChangeMoney(int money, AbstractMediator mediator)
    {
        mediator.BWin(money);
    }
}

/// <summary>
/// 抽象中介者类
/// </summary>
public abstract class AbstractMediator
{
    protected AbstractCardPartner A;
    protected AbstractCardPartner B;

    public AbstractMediator(AbstractCardPartner a, AbstractCardPartner b)
    {
        A = a;
        B = b;
    }

    public abstract void AWin(int money);
    public abstract void BWin(int money);
}

/// <summary>
/// 调用
/// </summary>
/// <param name="args"></param>
static void Main(string[] args)
{
    AbstractCardPartner A = new PartnerA();
    AbstractCardPartner B = new PartnerB();
    A.Money = 20;
    B.Money = 20;

    AbstractMediator mediator = new MediatorPater(A, B);
```

```
    // A赢了
    A.ChangeMoney(5, mediator);
    Console.WriteLine("A 现在的钱是: {0}", A.Money); // 应该是25
    Console.WriteLine("B 现在的钱是: {0}", B.Money); // 应该是15

    // B赢了
    B.ChangeMoney(10, mediator);
    Console.WriteLine("A 现在的钱是: {0}", A.Money); // 应该是15
    Console.WriteLine("B 现在的钱是: {0}", B.Money); // 应该是25

    Console.ReadLine();
}
```

在上面的实现代码中，抽象中介者类保存了两个抽象牌友类，如果新添加一个牌友类似时，此时就不得不去更改这个抽象中介者类。可以结合观察者模式来解决这个问题，即抽象中介者对象保存抽象牌友的类别，然后添加Register和UnRegister方法来对该列表进行管理，然后在具体中介者类中修改AWin和BWin方法，遍历列表，改变自己和其他牌友的钱数。这样的设计还是存在一个问题——即增加一个新牌友时，此时虽然解决了抽象中介者类不需要修改的问题，但此时还是不得不去修改具体中介者类，即添加CWin方法，我们可以采用状态模式来解决这个问题，关于状态模式的介绍将会在下一篇进行介绍。

中介者模式的优缺点

优点：

- 简化了对象之间的关系，将系统的各个对象之间的相互关系进行封装，将各个同事类解耦，使得系统变为松耦合。
- 提供系统的灵活性，使得各个同事对象独立而易于复用。

缺点：

- 中介者模式中，中介者角色承担了较多的责任，所以一旦这个中介者对象出现了问题，整个系统将会受到重大的影响。
- 新增加一个同事类时，不得不去修改抽象中介者类和具体中介者类，此时可以使用观察者模式和状态模式来解决这个问题。

中介者模式的适用场景

以下情况下可以考虑使用中介者模式：

- 一组定义良好的对象，现在要进行复杂的相互通信。
- 想通过一个中间类来封装多个类中的行为，而又不想生成太多的子类。

zml1234 4个月前 (11-23)