**三、观察者模式**

**观察者模式的定义：**

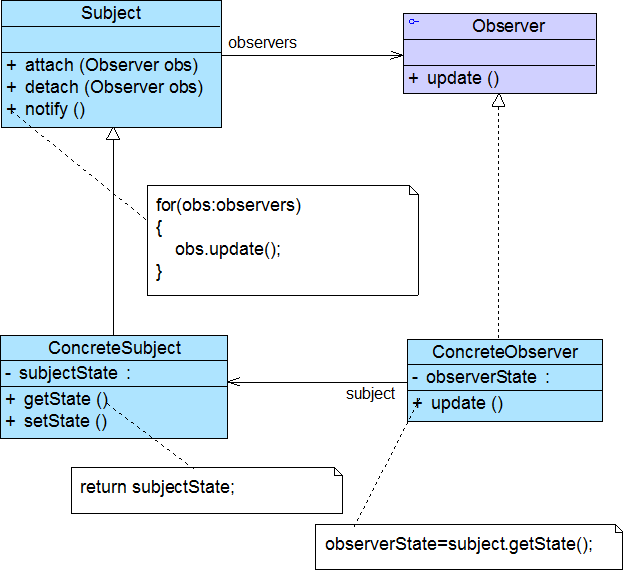
定义对象间的一种一对多依赖关系，使得每当一个对象状态发生改变时，其相关依赖对象皆得到通知并被自动更新。观察者模式又叫做发布-订阅（Publish/Subscribe）模式、模型-视图（Model/View）模式、源-监听器（Source/Listener）模式或从属者（Dependents）模式。

**观察者模式的结构：**

观察者模式包含如下角色

* Subject: 抽象目标
* ConcreteSubject: 具体目标
* Observer: 观察者接口
* ConcreteObserver: 具体观察者

观察者模式的基本作用原理图：



**观察者模式的几点特征：**

1. 观察者模式描述了如何建立对象与对象之间的依赖关系，如何构造满足这种需求的系统。
2. 这一模式中的关键对象是观察目标和观察者，一个目标可以有任意数目的与之相依赖的观察者，一旦目标的状态发生改变，所有的观察者都将得到通知。
3. 作为对这个通知的响应，每个观察者都将即时更新自己的状态，以与目标状态同步，这种交互也称为发布-订阅(publish-subscribe)。目标是通知的发布者，它发出通知时并不需要知道谁是它的观察者，可以有任意数目的观察者订阅它并接收通知。

**观察者模式的优缺点：**

观察者模式的优点：

1. 具体目标和具体观察者是松耦合关系。由于目标接口仅仅依赖于观察者接口，因此具体目标只是知道它的观察者是实现观察者接口的某个类的实例，但不需要知道具体是哪个类。同样，由于观察者仅仅依赖于目标接口，因此具体观察者只是知道它依赖的目标是实现目标接口的某个类的实例，但不需要知道具体是哪个类。
2. 观察模式满足“开-闭原则”。目标接口仅仅依赖于观察者接口，这样，我们就可以让创建具体目标的类也仅仅是依赖于观察者接口，因此如果增加新的实现观察者接口的类，不必修改创建具体目标的类的代码。同样，创建具体观察者的类仅仅依赖于目标接口，如果增加新的实现目标接口的类，也不必修改创建具体观察者类的代码。

观察者模式的缺点：

1. 如果一个观察目标对象有很多直接和间接的观察者的话，将所有的观察者都通知到会花费很多时间。
2. 如果在观察者和观察目标之间有循环依赖的话，观察目标会触发它们之间进行循环调用，可能导致系统崩溃。
3. 观察者模式没有相应的机制让观察者知道所观察的目标对象是怎么发生变化的，而仅仅只是知道观察目标发生了变化。

**一个观察者模式的实例：**

**Part1**: 目标 : Subject.java

public interface Subject

{

public void addObserver(Observer o);

public void deleteObserver(Observer o);

public void notifyObservers( );

}

**Part2**:观察者: Obsever.java

public interface Observer

{

public void hearTelephone(String heardMess);

}

**Part3**: 具体目标 SeekJobCenter.java

import java.util.ArrayList;

public class SeekJobCenter implements Subject

{

String mess;

boolean changed;

ArrayList<Observer> personList;

SeekJobCenter( ){

personList=new ArrayList<Observer>( );

mess="";

changed=false;

}

**Part4**: 具体观察者\_1 UniversityStudent.java

import java.io.\*;

public class UniverStudent implements Observer{

Subject subject;

File myFile;

UniverStudent(Subject subject,String fileName){

this.subject=subject;

subject.addObserver(this); //使当前实例成为subject所引用的具体主题的观察者

myFile=new File(fileName);

}

public void hearTelephone(String heardMess){

try{ RandomAccessFile out=new RandomAccessFile(myFile,"rw");

out.seek(out.length());

byte [] b=heardMess.getBytes();

out.write(b); //更新文件中的内容

System.out.print("我是一个大学生,");

System.out.println("我向文件"+myFile.getName()+"写入如下内容:");

System.out.println(heardMess);

}

catch(IOException exp){

System.out.println(exp.toString());

}

}

}

**中介者模式**

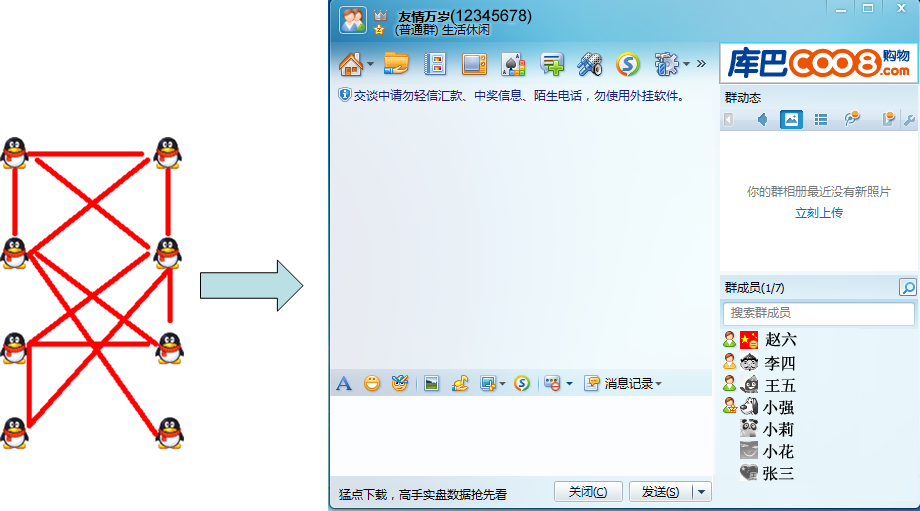
**中介者模式的来源**

* 在用户与用户直接聊天的设计方案中，用户对象之间存在很强的关联性，将导致系统出现如下问题：

1. 系统结构复杂：对象之间存在大量的相互关联和调用，若有一个对象发生变化，则需要跟踪和该对象关联的其他所有对象，并进行适当处理
2. 对象可重用性差：由于一个对象和其他对象具有很强的关联，若没有其他对象的支持，一个对象很难被另一个系统或模块重用，这些对象表现出来更像一个不可分割的整体，职责较为混乱。

* 在面向对象的软件设计与开发过程中，根据“单一职责原则”，我们应该尽量将对象细化，使其只负责或呈现单一的职责。
* 对于一个模块，可能由很多对象构成，而且这些对象之间可能存在相互的引用，为了减少对象两两之间复杂的引用关系，使之成为一个松耦合的系统，我们需要使用中介者模式，这就是中介者模式的模式动机。

**中介者模式的一个经典案例：**



**中介者模式的定义**

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。

**中介者模式结构**

中介者模式包含如下角色：

* Mediator: 抽象中介者
* ConcreteMediator: 具体中介者
* Colleague: 抽象同事类
* ConcreteColleague: 具体同事类

中介者模式的基本结构：



**中介者模式的基本特征**

* 中介者模式可以使对象之间的关系数量急剧减少：



* 中介者承担两方面的职责：

1. 中转作用（结构性）：通过中介者提供的中转作用，各个同事对象就不再需要显式引用其他同事，当需要和其他同事进行通信时，通过中介者即可。该中转作用属于中介者在结构上的支持。
2. 协调作用（行为性）：中介者可以更进一步的对同事之间的关系进行封装，同事可以一致地和中介者进行交互，而不需要指明中介者需要具体怎么做，中介者根据封装在自身内部的协调逻辑，对同事的请求进行进一步处理，将同事成员之间的关系行为进行分离和封装。该协调作用属于中介者在行为上的支持。

**中介者模式的优缺点**

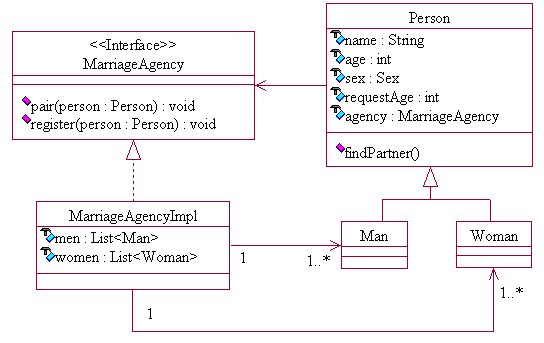
中介者模式的优点

1. 简化了对象之间的交互
2. 将各同事解耦，减少子类生成
3. 可以简化各同事类的设计和实现

中介者模式的缺点

* 在具体中介者类中包含了同事之间的交互细节，可能会导致具体中介者类非常复杂，使得系统难以维护。

**一个中介者模式的实例：**



* 抽象中介者 : MarriageAgency.java

public interface MarriageAgency {

void pair(Person person); // 为person配对

void register(Person person); // 注册会员

}

* 抽象同事类: Person.java

public abstract class Person {

String name; // 姓名

int age; // 年龄

Sex sex; // 性别

int requestAge; // 要求对象的年龄。对对象只有这一个要求

MarriageAgency agency; // 婚姻中介

public Person(String name, int age, Sex sex, int requestAge,

MarriageAgency agency) {

this.name = name;

……

agency.register(this); // 注册会员

}

// 寻找对象

public void findPartner() {

agency.pair(this);

}

}

* 具体中介者 : MarriageAgencyImpl.java

public class MarriageAgencyImpl implements MarriageAgency {

List<Man> men = new ArrayList<Man>(); // 男会员

List<Woman> women = new ArrayList<Woman>(); // 女会员

public void register(Person person) {

if (person.sex == Sex.MALE)

men.add((Man) person);

else if (person.sex == Sex.FEMALE)

women.add((Woman) person);

}

public void pair(Person person) {

if (person.sex == Sex.MALE) {

for (Woman w : women)

if (w.age == person.requestAge) {

System.out.println(person.name + "和" + w.name + "配对成功");

return;

}

} else if (person.sex == Sex.FEMALE) {

for (Man m : men)

if (m.age == person.requestAge) {

System.out.println(person.name + "和" + m.name + "配对成功");

return;

}

}

System.out.println("没有为" + person.name + "找到合适的对象");

}

}

* 具体同事类1 : Man.java

public class Man extends Person {

public Man(String name, int age, int requestAge,

MarriageAgency agency) {

super(name, age, Sex.MALE, requestAge, agency);

}

}

* 测试类: Test.java

public class Test {

public static void main(String[] args) {

MarriageAgency agency = new MarriageAgencyImpl();

Person m1 = new Man("John", 20, 18, agency);

Person m2 = new Man("Mike", 27, 25, agency);

Person w1 = new Woman("Mary", 25, 27, agency);

Person w2 = new Woman("Jane", 20, 22, agency);

m1.findPartner();

m2.findPartner();

}