中介者模式

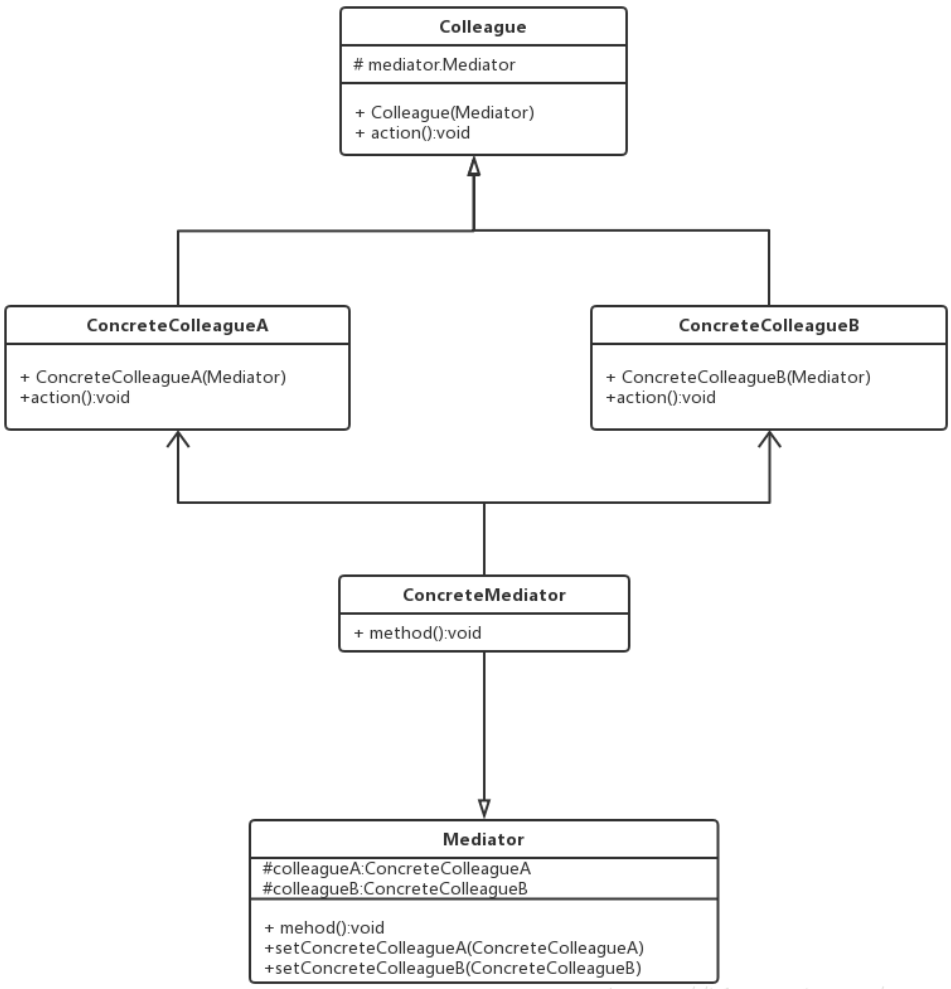
1.中介者模式的定义

中介者模式包装了一系列对象相互作用的方式，是的这些对象不必相互明显作用。从而使它们可以松散耦合。当某些对象之间的作用发生变化时，不会立即影响其它的一些对象之间的作用。保证这些作用可以彼此独立的变化。中介者模式将多对多的相互作用转化为一对多的相互作用。终结者模式将对象的行为和写作抽象化，把对象在小尺度的行为上与其他对象的相互作用分开处理。

2.中介者模式的使用场景

当对象之间爱你的交互操作很多且每个对象的行为操作都依赖彼此时，为防止在修改一个对象的行为时，同时涉及修改很多其他对象的行为，可采用中介者模式，来解决紧耦合问题。该模式将对象之间的多对多关系变成一对多关系，终结者对象将系统从网状结构变成调停者为中心的星型结构，达到降低系统的复杂性，提高可扩展性的作用。

3.终结者模式的UML图



角色介绍：

- Mediator:抽象中介者角色，定义了同时对象达到中介者对象的接口，，一般以抽象类的方式实现。
- ConcreteMediator:具体中介者角色，继承于抽象中介者，实现了父类定义的方法，他从具体的同事对象接受消息，向具体同事对象发出命令。
- Colleague：抽象同时类角色，定义了中介者对象的接口，他只知道中介者而不知道其他的同时对象。
- ConcreteColleagueA/B:具体同事类角色，继承于抽象通识类，每个具体通识类都知道本身在小范围内的行为，而不知道他在大范围内的目的。

4.中介者模式的简单实现

中介者模式的例子很多，其实我们每天使用的电脑就是一个中介者模式应用的例子，以电脑主机为例，我们都知道电脑主机部分构成主要分为几块：CPU、内存、显卡、IO设备，一般来说一台电脑有了前两块既可以运行启动了，当然，如果你要连接显示器显示画面，那么还需要加上显卡，如果你还需要存储数据什么的，那还需要加上IO设备，但是，在本节中这些部分都不是重要，因为他们分隔开来就是一个普通的零部件而已，我们需要一样东西将这些零部件都整合起来变成一个完整的整体，这个东西就是主板，在这里主板就起到中介者的作用，连接CPU、内存、显卡和IO设备等，任何亮膜款之间的通信都会经过主板去协调，这里读取光盘为例，来看看主板是如何充当着中介者角色的，首先还是定义一个抽象的中介者：

```
package mediatorpattern;
//抽象中介者
public abstract class Mediator {
/*
 * 同事对象改变时通知中介者的方法
 * 在同事对象改变时由中介者去通知其他的同事对象
 */
        public abstract void changed(Colleague c);
}
```

抽象中介者只是定义了一个抽象接口方法，具体的通识类通过该方法来通知中介者自身的状态改变。而具体的中介者这里就是指主板，由他负责联系各个具体同事类，也就是CPU、内存、显卡、IO设备等。

```
package mediatorpattern;

import java.util.logging.Handler;

public class MainBoard extends Mediator {

        private CDDevice cdDevice;// 光驱设备
        private CPU cpu;// CPU
        private SoundCard soundCard;// 声卡设备
        private GraphicsCard graphicsCard;// 显卡设备

        @Override
        public void changed(Colleague c) {
```

```
            // 如果是光驱读取了数据
                                        if (c == cdDevice) {
                HandlerCD((CDDevice) c);
            } else if (c == cpu) {
                // 如果CPU处理完数据
                HandlerCPU((CPU) c);
            }
        }

        /*
         * 处理光驱读取数据后与其他设备的交互
         */
        private void HandlerCD(CDDevice c) {
            cpu.decodeData(cdDevice.read());

        }

        /*
         * 处理CPU读取数据后与其他设备的交互
         */
        private void HandlerCPU(CPU c) {
            soundCard.soundPlay(cpu.getDataSound());
            graphicsCard.videoPlay(cpu.getDataVideo());
        }

        /*
         * 设置CD设备
         */
        public void setCDDevice(CDDevice cdDevice) {
            this.cdDevice = cdDevice;
        }

        /*
         * 设置CPU
         */
        public void setCpu(CPU cpu) {
            this.cpu = cpu;
        }

        /*
         * 设置声卡设备
         */
        public void setSoundCard(SoundCard soundCard) {
            this.soundCard = soundCard;
        }

        /*
         * 设置显卡设备
         */
        public void setGraphicsCard(GraphicsCard graphicsCard) {
```

```
                this.graphicsCard = graphicsCard;        }
    }
```

抽象同事类里只有一个抽象中介者的引用，我们在构造方法中为其赋值。

```
package mediatorpattern;
/*
 * 抽象同事类
 */
public abstract class Colleague {
        protected Mediator mediator;// 每一同事都该知道其中介者

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

接下来就是各个具体的零部件了，首先是CPU，其负责从主板传递来音、视频数据的解码。

```
package mediatorpattern;

public class CPU extends Colleague {
        private String dataVideo, dataSound;// 视频和音频数据

        public CPU(Mediator mediator) {
                super(mediator);
        }

        /*
         * 获取视频数据
         */
        public String getDataVideo() {
                return dataVideo;
        }

        /*
         * 获取音频数据
         */
        public String getDataSound() {
                return dataSound;
        }

        /*
         * 解码数据
         */
        public void decodeData(String data) {
```

```
                // 分割音、视频数据
                String[] tmp = data.split(",");
                // 解析音、视频数据
                dataVideo = tmp[0];
                dataSound = tmp[1];

                // 告诉中介者自身状态改变
                mediator.changed(this);
        }
}
```

而CD设备则负责读取光盘的数据并将数据提供给主板

```
package mediatorpattern;

/*
 * 光驱同事
 */
public class CDDevice extends Colleague {
        private String data;// 视频数据

        public CDDevice(Mediator mediator) {
                super(mediator);
        }
        /*
         * 读取视频数据
         */
        public String read(){
                return data;
        }
        /*
         * 加载视频数据
         */
        public void load(){
                //实际情况中视频数据与音频数据都在一个数据流中
                data="视频数据,音频数据";
                //通知中介者 也就是主板数据改变
                mediator.changed(this);
        }
 }
```

显卡和声卡分别用来播放视频和音频，其逻辑相对来说简单。

```
package mediatorpattern;

public class GraphicsCard extends Colleague{

        public GraphicsCard(Mediator mediator) {
```

```
                super(mediator);
                                        }
/*
 * 播放视频
 */
        public void videoPlay(String data){
                System.out.println("视频: "+data);
        }
}
```

```
package mediatorpattern;

public class SoundCard extends Colleague{

        public SoundCard(Mediator mediator) {
                super(mediator);
        }
/*
 * 播放音频
 */
        public void soundPlay(String d){
                System.out.println("音频:  "+d);
        }
}
```

最后通过一个客户端来模拟电脑播放电影的效果。

```
package mediatorpattern;

public class Client {
        public static void main(String[] args) {
                // 构造主板对象
                MainBoard mediator = new MainBoard();

                // 分别构造各个零部件
                CDDevice cd = new CDDevice(mediator);
                CPU cpu = new CPU(mediator);
                GraphicsCard vc = new GraphicsCard(mediator);
                SoundCard sc = new SoundCard(mediator);

                // 将各个零部件安装到主板
                mediator.setCDDevice(cd);
                mediator.setCpu(cpu);
                mediator.setGraphicsCard(vc);
                mediator.setSoundCard(sc);

                // 完成后就可以开始放片
```

```
                cd.load();          }

    }
```

输出结果如下：

```
音频：  音频数据
视频：视频数据
```

从上述程序演示可以明白，中介者模式就是用来协调多个对象之间的交互的，就像上面示例中的主板，如果没有主板这个中介者，那么电脑里的每一个零部件都要与其他零部件建立关联，比如CPU要与内存交互、CPU要与显卡交互、CPU还要与IO设备交互等，这么一来就会构成一个错综复杂的网状图，而中介者模式的出现则将这一错综复杂的网状图变成一个结构清晰的星形图，其中心就是中介者。

5.总结

在面向对象的编程语言里，一个类必然会与其他类产生依赖关系，如果这种依赖关系入网状般错综复杂，那么必然会影响我们的代码逻辑以及执行效率，是当地使用中介者模式可以对这种依赖关系进行解耦使逻辑结构清晰，但是，如果及各类间的依赖关并不复杂，使用中介者模式反而会使得原本不复杂的逻辑结构变得复杂，所以，我们在决定hi用中介者模式之前要多方考虑、权衡利弊。

源码资源

《Android源码设计模式解析与实战 17章》