# 行为型模式（二）

2020年5月12日 11:10

行为型设计模式（二）

作业要求：

作业要求：

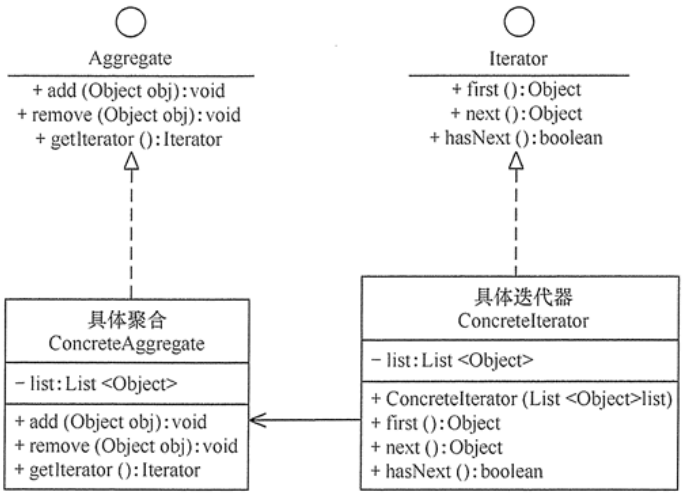
1、结合实例，绘制迭代器模式实例结构图（类图），用面向对象编程语言实现该模式，并对模式进行分析。

2、结合实例，绘制中介者模式实例结构图（类图），用面向对象编程语言实现该模式，并对模式进行分析。

3、某教务管理系统中一个班级（Class）包含多个学生（Student），使用Java内置的迭代器实现对学生信息的遍历，要求按学生的年龄由大到小的次序输出学生信息。用java语言模拟实现过程。

- **作业与笔记github地址**：
  https://github.com/baobaotql/CCNU_DesignPattern

- **Q1:结合实例，绘制迭代器模式实例结构图（类图），用面向对象编程语言实现该模式，并对模式进行分析。**
  **A1：**
  - **迭代器定义**
    提供一个对象来顺序访问聚合对象中的一系列数据，而不暴露聚合对象的内部表示。迭代器模式是一种对象行为型模式。
  - **优点**
    - 访问一个聚合对象的内容而无须暴露它的内部表示。
    - 遍历任务交由迭代器完成，这简化了聚合类。
    - 它支持以不同方式遍历一个聚合，甚至可以自定义迭代器的子类以支持新的遍历。
    - 增加新的聚合类和迭代器类都很方便，无须修改原有代码。
    - 封装性良好，为遍历不同的聚合结构提供一个统一的接口。
  - **缺点**
    - 增加了类的个数，这在一定程度上增加了系统的复杂性。
  - **模式结构**
    - 抽象聚合（Aggregate）角色：定义存储、添加、删除聚合对象以及创建迭代器对象的接口。
    - 具体聚合（ConcreteAggregate）角色：实现抽象聚合类，返回一个具体迭代器的实例。
    - 抽象迭代器（Iterator）角色：定义访问和遍历聚合元素的接口，通常包含 hasNext()、first()、next() 等方法。
    - 具体迭代器（ConcreteIterator）角色：实现抽象迭代器接口中所定义的方法，完成对聚合对象的遍历，记录遍历的当前位置。
    - **模式结构图**




- **模式分析**

  迭代器模式是通过将聚合对象的遍历行为分离出来，抽象成迭代器类来实现的，其目的是在不暴露聚合对象的内部结构的情况下，让外部代码透明地访问聚合的内部数据。

- **模式结构代码**

```
package iterator;
import java.util.*;
public class IteratorPattern
{
    public static void main(String[] args)
    {
        Aggregate ag=new ConcreteAggregate();
        ag.add("东北大学");
        ag.add("华中师范");
        ag.add("工程中心");
        System.out.print("聚合的内容有：");
        Iterator it=ag.getIterator();
        while(it.hasNext())
        {
            Object ob=it.next();
            System.out.print(ob.toString()+"\t");
        }
        Object ob=it.first();
        System.out.println("\nFirst: "+ob.toString());
    }
}
//抽象聚合
interface Aggregate
{
    public void add(Object obj);
    public void remove(Object obj);
    public Iterator getIterator();
}
//具体聚合
class ConcreteAggregate implements Aggregate
{
    private List<Object> list=new ArrayList<Object>();
    public void add(Object obj)
    {
        list.add(obj);
    }
    public void remove(Object obj)
    {
        list.remove(obj);
```

```
        }
        public Iterator getIterator()
        {
            return(new ConcreteIterator(list));
        }
    }
    //抽象迭代器
    interface Iterator
    {
        Object first();
        Object next();
        boolean hasNext();
    }
    //具体迭代器
    class ConcreteIterator implements Iterator
    {
        private List<Object> list=null;
        private int index=-1;
        public ConcreteIterator(List<Object> list)
        {
            this.list=list;
        }
        public boolean hasNext()
        {
            if(index<list.size()-1)
            {
                return true;
            }
            else
            {
                return false;
            }
        }
        public Object first()
        {
            index=0;
            Object obj=list.get(index);;
            return obj;
        }
        public Object next()
        {
            Object obj=null;
            if(this.hasNext())
            {
                obj=list.get(++index);
            }
            return obj;
        }
    }
```

- **Q2：结合实例，绘制中介者模式实例结构图（类图），用面向对象编程语言实现该模式，并对模式进行分析。**

  **A2：**

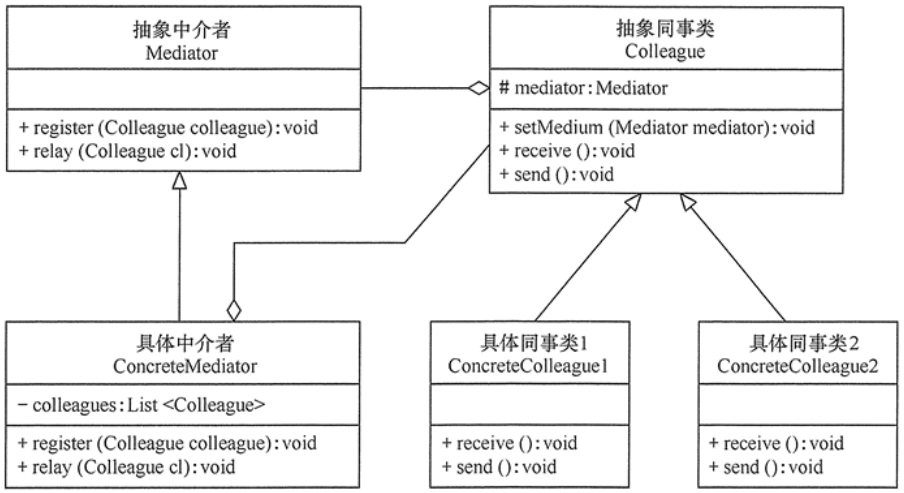
  - **中介者定义**

    定义一个中介对象来封装一系列对象之间的交互，使原有对象之间的耦合松散，且可以独立地改变它们之间的交互。中介者模式又叫调停模式，它是迪米特法则的典型应用。

  - **优点**
    - 降低了对象之间的耦合性，使得对象易于独立地被复用。
    - 将对象间的一对多关联转变为一对一的关联，提高系统的灵活性，

使得系统易于维护和扩展。

- **缺点**
  - 当同事类太多时，中介者的职责将很大，它会变得复杂而庞大，以至于系统难以维护
- **模式结构**
  - 抽象中介者（Mediator）角色：它是中介者的接口，提供了同事对象注册与转发同事对象信息的抽象方法。
  - 具体中介者（ConcreteMediator）角色：实现中介者接口，定义一个 List 来管理同事对象，协调各个同事角色之间的交互关系，因此它依赖于同事角色。
  - 抽象同事类（Colleague）角色：定义同事类的接口，保存中介者对象，提供同事对象交互的抽象方法，实现所有相互影响的同事类的公共功能。
  - 具体同事类（Concrete Colleague）角色：是抽象同事类的实现者，当需要与其他同事对象交互时，由中介者对象负责后续的交互。
  - **模式结构图**



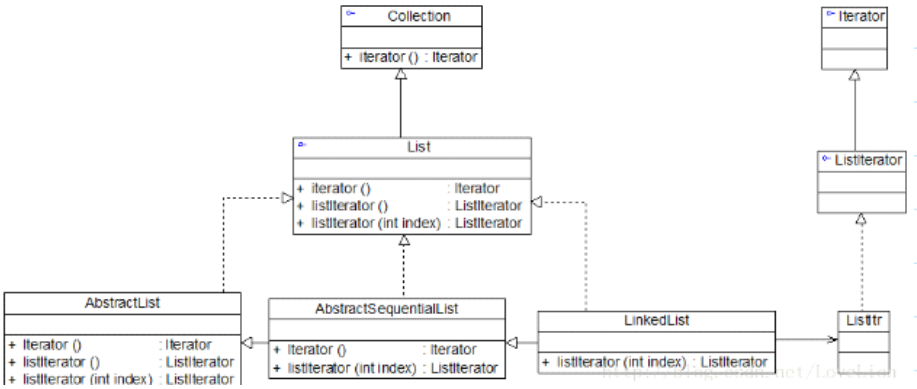

  - **模式代码**

```
package mediator;
import java.util.*;
public class MediatorPattern
{
    public static void main(String[] args)
    {
        Mediator md=new ConcreteMediator();
        Colleague c1,c2;
        c1=new ConcreteColleague1();
        c2=new ConcreteColleague2();
        md.register(c1);
        md.register(c2);
        c1.send();
        System.out.println("--------------");
        c2.send();
    }
}
//抽象中介者
abstract class Mediator
{
    public abstract void register(Colleague colleague);
    public abstract void relay(Colleague cl); //转发
}
//具体中介者
```

```
class ConcreteMediator extends Mediator
{
    private List<Colleague> colleagues=new ArrayList<Colleague>();
    public void register(Colleague colleague)
    {
        if(!colleagues.contains(colleague))
        {
            colleagues.add(colleague);
            colleague.setMedium(this);
        }
    }
    public void relay(Colleague cl)
    {
        for(Colleague ob:colleagues)
        {
            if(!ob.equals(cl))
            {
                ((Colleague)ob).receive();
            }
        }
    }
}
//抽象同事类
abstract class Colleague
{
    protected Mediator mediator;
    public void setMedium(Mediator mediator)
    {
        this.mediator=mediator;
    }
    public abstract void receive();
    public abstract void send();
}
//具体同事类
class ConcreteColleague1 extends Colleague
{
    public void receive()
    {
        System.out.println("具体同事类1收到请求。");
    }
    public void send()
    {
        System.out.println("具体同事类1发出请求。");
        mediator.relay(this); //请中介者转发
    }
}
//具体同事类
class ConcreteColleague2 extends Colleague
{
    public void receive()
    {
        System.out.println("具体同事类2收到请求。");
    }
    public void send()
    {
        System.out.println("具体同事类2发出请求。");
        mediator.relay(this); //请中介者转发
    }
}
```

- **Q3:某教务管理系统中一个班级（Class）包含多个学生（Student），使用Java内置的迭代器实现对学生信息的遍历，要求按学生的年龄由大到小的次序输出学生信息。用java语言模拟实现过程。**

**A3:**




- **学生类**

```
package gjIteratorPattern;

public class Student implements Comparable<Student>{
    private String name;
    private int age;
    public Student(String name, int age) {
        super();
        this.name = name;
        this.age = age;
    }
    public int compareTo(Student stu){
        if(this.age > stu.age){
            return -1;
        }else if(this.age < stu.age){
            return 1;
        }
        return 0;
    }
    public String getName() {
        return name;
    }
    public void setName(String name) {
        this.name = name;
    }
    public int getAge() {
        return age;
    }
    public void setAge(int age) {
        this.age = age;
    }
}
```

- **Java内置迭代器**

```
package gjIteratorPattern;

import java.util.ArrayList;
import java.util.Collection;
import java.util.Collections;
import java.util.Iterator;
import java.util.List;

public class JavaIterator {
    List<Student> slist=null;
    public JavaIterator(){
        Student[] stu=new Student[5];
```

```
            slist=new ArrayList<Student>();
            stu[0]=new Student("张三", 32);
            stu[1]=new Student("李四", 25);
            stu[2]=new Student("王五", 21);
            stu[3]=new Student("赵六", 38);
            stu[4]=new Student("周七", 26);
            for(int i=0;i<5;i++){
                slist.add(stu[i]);
            }
        }
        public void display(){
            Iterator<Student> t=slist.iterator();
            System.out.println("遍历获得的原始数据: ");
            while(t.hasNext()){
                Student student=t.next();
                System.out.println("姓名: "+student.getName()+"今
年"+student.getAge()+"岁");
            }
            Collections.sort(slist);
            Iterator<Student> it=slist.iterator();
            System.out.println("=====================================");
            System.out.println("按年龄从大到小排序: ");
            while(it.hasNext()){
                Student student=it.next();
                System.out.println("姓名: "+student.getName()+"今
年"+student.getAge()+"岁");
            }
        }


    }
```

- **客户端**

```
package gjIteratorPattern;
/**
 * 迭代器模式客户端
 * @author baobao
 *
 */
public class Client {
    public static void main(String[] args) {
        JavaIterator JavaIterator= new JavaIterator();
        JavaIterator.display();
    }
}
```