迭代器模式

2020年2月16日 12:18

1. 概念

提供了一个迭代器，通过迭代器中所提供的方法可以顺序访问一个聚合对象中的各个元素，而又不暴露该对象的内部表示。

2. UML结构图

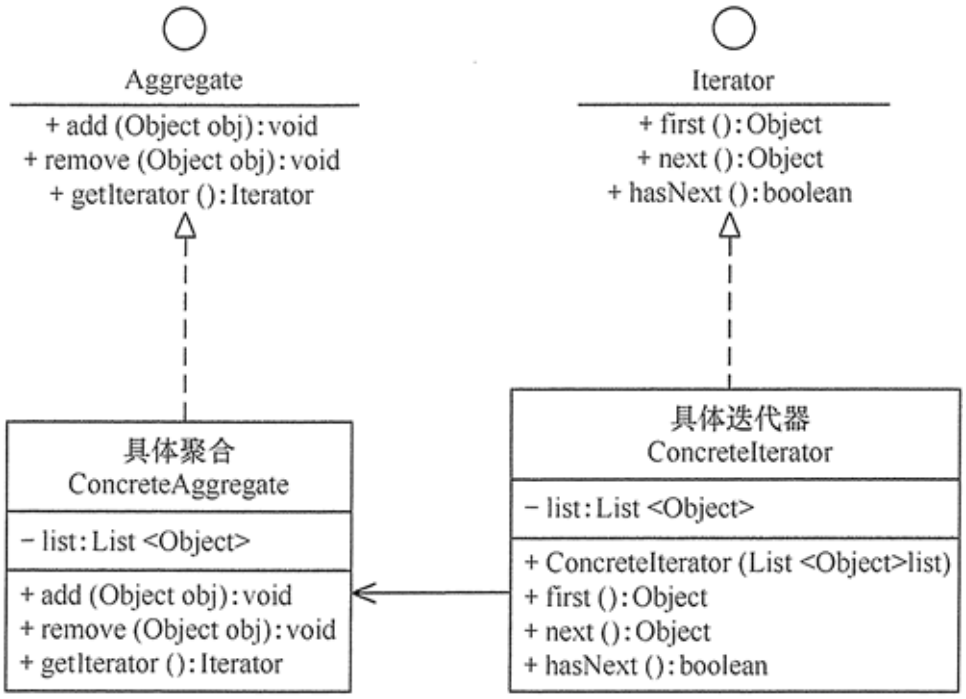

图2.1 迭代器模式UML结构图

1. 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。
2. 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。
3. 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。
4. 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。

3. 迭代器模式实例

创建一个课程的聚合对象，使用迭代器模式将聚合对象和其的遍历行文进行分离。

1. 被聚合类

```
public class Course {
    private String name;
    public Course(String name) {
        this.name = name;
    }

    public String getName() {
        return name;
    }

    @Override
    public String toString() {
        return "Course{" +
                "name='" + name + '\'' +
```

```
                '}';
    }
}
```

2. 聚合对象接口

```
public interface CourseAggregate {
    void addCourse(Course course);
    void removeCourse(Course course);

    CourseIterator getCourseIterator();
}
```

3. 具体聚合实现类

```
public class CourseAggregateImpl implements CourseAggregate {
    private List courseList;

    public CourseAggregateImpl() {
        this.courseList = new ArrayList();
    }

    @Override
    public void addCourse(Course course) {
        courseList.add(course);
    }

    @Override
    public void removeCourse(Course course) {
        courseList.remove(course);
    }

    // 获取课程迭代器
    @Override
    public CourseIterator getCourseIterator() {
        return new CourseIteratorImpl(courseList);
    }
}
```

4. 迭代器接口

```
public interface CourseIterator {
    Course nextCourse();
    boolean isLastCourse();
}
```

5. 迭代器实现类

```
public class CourseIteratorImpl implements CourseIterator {
    private List courseList;
    private int position;
    private Course course;
    public CourseIteratorImpl(List courseList) {
        this.courseList = courseList;
    }

    @Override
    public Course nextCourse() {
        System.out.println("返回课程,位置是:" + position);
        course = (Course) courseList.get(position);
        position++;
        return course;
    }

    @Override
    public boolean isLastCourse() {
        if (position < courseList.size()){
            return false;
        }
        return true;
    }
}
```

6. 客户端

```
public class iterator {
    @Test
    public void iteratorTest(){
        Course designPatternCourse = new Course("java 设计模式");
        Course FECourse = new Course("前端课程");
        Course javaCourse = new Course("java课程");

        // 创建Course对象集合类
        CourseAggregate courseAggregate = new CourseAggregateImpl();
        courseAggregate.addCourse(designPatternCourse);
        courseAggregate.addCourse(FECourse);
        courseAggregate.addCourse(javaCourse);

        System.out.println("===拥有课程列表===");
```

```
        // 输出迭代器元素
        printCourse(courseAggregate);

        // 删除课程
        System.out.println("===删除FE课程===");
        courseAggregate.removeCourse(FECourse);
        printCourse(courseAggregate);
    }

    public void printCourse(CourseAggregate courseAggregate){
        // 获取迭代器
        CourseIterator courseIterator = courseAggregate.getCourseIterator();
        while (!courseIterator.isLastCourse()){
            System.out.println(courseIterator.nextCourse().toString());
        }
    }
}
```

7. 结果

```
返回课程,位置是:0
Course{name='java 设计模式'}
返回课程,位置是:1
Course{name='前端课程'}
返回课程,位置是:2
Course{name='java课程'}
===删除FE课程===
返回课程,位置是:0
Course{name='java 设计模式'}
返回课程,位置是:1
Course{name='java课程'}
```

4. 优/缺点

1. 优点

- 分离了聚合对象和其遍历行为，简化了集合对象。
- 访问一个聚合对象而不需要暴露它的内部表示。
- 支持以不用的方式遍历一个聚合对象，增加相应的迭代器即可，符合开闭原则。
- 封装性良好，为遍历不用的聚合结构提供了一个统一的接口。

2. 缺点

类的个数成对增加，从而增加了系统复杂性。

5. 适用场景

1. 访问一个集合对象的内容而无需暴露它的内部表示。
2. 为遍历不同集合结构提供一个统一的接口。
3. 当需要为聚合对象提供多种遍历方式时。

6. 扩展

迭代器模式常常与组合模式结合起来使用，在对组合模式中的容器构件进行访问时，经常将迭代器潜藏在组合模式的容器构成类中。当然，也可以构造一个外部迭代器来对容器构件进行访问，其结构图如下所示。

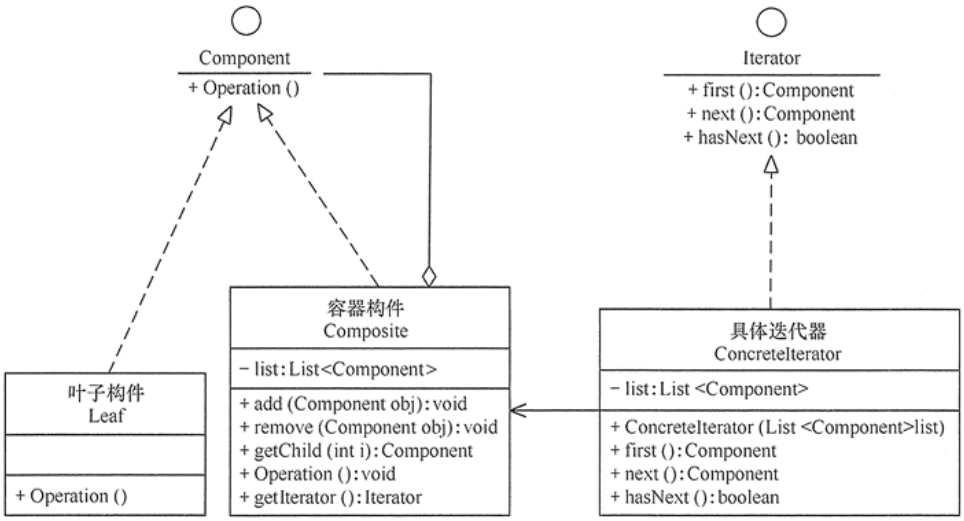

图6.1 迭代器模式和组合模式的结合

7. 参考资料

[1] http://c.biancheng.net/view/1395.html