概念

访问者模式是一种将数据结构与结构中元素的操作分离的一种设计模式

使用场景

1. 数据结构比较稳定，但经常需要在此数据结构上定义新的操作来访问结构中的元素
2. 需要对一个结构中的元素进行一些很多不同的并且不相关的操作操作，但是不想污染数据结构

角色介绍

- **Visitor**：接口或者抽象类，定义了对每个 Element 访问的行为，它的参数就是被访问的元素，它的方法个数理论上与元素的个数是一类型要稳定，如果经常添加、移除元素类，必然会导致频繁地修改 Visitor 接口，如果出现这种情况，则说明不适合使用访问者模式。
- **ConcreteVisitor**：具体的访问者，它需要给出对每一个元素类访问时所产生的具体行为。
- **Element**：元素接口或者抽象类，它定义了一个接受访问者（accept）的方法，其意义是指每一个元素都要可以被访问者访问。
- **ElementA、ElementB**：具体的元素类，它提供接受访问的具体实现，而这个具体的实现，通常情况下是使用访问者提供的访问该元
- **ObjectStructure**：定义当中所提到的对象结构，对象结构是一个抽象表述，它内部管理了元素集合，并且可以迭代这些元素提供访

代码

年底，CEO和CTO开始评定员工一年的工作绩效，员工分为工程师和经理，CTO关注工程师的代码量、经理的新产品数量；CEO关注的是工量。

由于CEO和CTO对于不同员工的关注点是不一样的，这就需要对不同员工类型进行不同的处理。访问者模式此时可以派上用场了

```
/**
 * 元素
 */
public abstract class Element {
    protected String name;
    protected int kpi;

    public Element(String name){
        this.name = name;
         kpi = new Random().nextInt(10);
     }

     // 接受访问
     abstract void accept(Visitor visitor);
```

```java
}

/**
 * 工程师
 */
public class Engineer extends Element {

    private int codeLines = new Random().nextInt(10 * 10000);

    public Engineer(String name) {
        super(name);
    }

    // 工程师一年的代码数量
    public int getCodeLines() {
        return codeLines;
    }

    @Override
    void accept(Visitor visitor) {
        visitor.visit(this);
    }
}

/**
 * 产品经理
 */
public class ProductManager extends Element {

    private int productNumber = new Random().nextInt(10);

    public ProductManager(String name) {
        super(name);
    }

    @Override
    void accept(Visitor visitor) {
        visitor.visit(this);
    }

    // 一年的产品数
    public Integer getProductNumber(){
        return productNumber;
    }
}

/**
 * 访问者接口
 */
public interface Visitor {
    /**
     * 访问工程师
     * @param engineer
     */
```

```
    void visit(Engineer engineer);

    /**
     * 访问产品经理
     * @param productManager
     */
    void visit(ProductManager productManager);
}

public class CEOVisitor implements Visitor {
    @Override
    public void visit(Engineer engineer) {
        System.out.println("工程师 : " + engineer.name + " 的考核情况 ====>  " +
                                    "kpi : " + engineer.kpi + ",  codelines : " + engineer.getCode
    }

    @Override
    public void visit(ProductManager productManager) {
        System.out.println("产品经理: " + productManager.name + " 的考核情况 ====>  " +
                                    "kpi : " + productManager.kpi + ",  productNumber : " + produc
    }
}

public class CTOVisitor implements Visitor {

    @Override
    public void visit(Engineer engineer) {
        System.out.println("工程师: " + engineer.name + " ,  代码行数: " + engineer.getCodeLines());
    }

    @Override
     public void visit(ProductManager productManager) {
         System.out.println("产品经理: " + productManager.name + ", 产品数量: " + productManager.getPr
     }
}

/**
 * 元素结构：用来存储 工程师 和 产品经理，以及一个固定方法（展示员工业务报表）
 */
public class ObjectStructure {

    private List<Element> elements = new LinkedList<>();

    public ObjectStructure(){
        elements.add(new Engineer("A"));
        elements.add(new Engineer("B"));
        elements.add(new Engineer("C"));
        elements.add(new Engineer("D"));

        elements.add(new ProductManager("甲"));
        elements.add(new ProductManager("乙"));
        elements.add(new ProductManager("丙"));
        elements.add(new ProductManager("丁"));
    }
```

```
123
124     public void showReport(Visitor visitor){
125         for (Element element : elements) {
126             element.accept(visitor);
127         }
128     }
129 }
130
131 public class Main {
132
133     public static void main(String[] args) {
134         ObjectStructure objectStructure = new ObjectStructure();
135
136         CEOVisitor ceo = new CEOVisitor();
137         System.out.println("ceo考核");
138         objectStructure.showReport(ceo);
139         System.out.println();
140
141         CTOVisitor cto = new CTOVisitor();
142         System.out.println("cto考核");
143         objectStructure.showReport(cto);
144     }
145 }
```

ceo和cto对员工的考核维度是不一样的。那么ceo和cto对元素的访问方式就不一样。所以这里采用的访问者模式。

如果不用访问者模式，那么代码大致如下:

```
1  public void showReport2(Visitor visitor){
2      for (Element element : elements) {
3          if (visitor instanceof CEOVisitor){
4              CEOVisitor ceoVisitor = (CEOVisitor) visitor;
5              // doSomething
6          }else if (visitor instanceof CTOVisitor){
7              CTOVisitor ctoVisitor = (CTOVisitor) visitor;
8              // doSomething
9          }
10     }
11 }
```

if-else 逻辑的嵌套以及类型的强制转换，难以扩展和维护，当类型较多时，这个方法会越来越复杂

总结

我们要根据具体情况来评估是否适合使用访问者模式，例如，我们的对象结构是否足够稳定，是否需要经常定义新的操作，使用访问者模式的代码变得更复杂