课程目标

1、掌握访问者模式的应用场景。

1、了解访问者模式的双分派。

4、了解访问者模式的优、缺点。

内容定位

1、访问者模式被称为最复杂的设计模式。

访问者模式

访问者模式（Visitor Pattern）是一种将数据结构与数据操作分离的设计模式。是指封装一些作用于某种数据结构中的各元素的操作，它可以在不改变数据结构的前提下定义作用于这些元素的新的操作。属于行为型模式。

```
原文：Represent an operation to be performed on the elements of an object structure. Visitor lets you define a new operation without changing the classes of the elements on which it operates.
```

访问者模式被称为最复杂的设计模式，并且使用频率不高，设计模式的作者也评价为：大多情况下，你不需要使用访问者模式，但是一旦需要使用它时，那就真的需要使用了。访问者模式的基本思想是，针对系统中拥有固定类型数的对象结构（元素），在其内提供一个 accept() 方法用来接受访问者对象的访问。不同的访问者对同一元素的访问内容不同，使得相同的元素集合可以产生不同的数据结果。accept()方法可以接收不同的访问者对象，然后在内部将自己（元素）转发到接收到的访问者对象的 visit()方法内。访问者内部对应类型的 visit()方法就会得到回调执行，对元素进行操作。也就是通过两次动态分发（第一次是对访问者的分发 accept()方法，

第二次是对元素的分发 visit()方法），才最终将一个具体的元素传递到一个具体的访问者。如此一来，就解耦了数据结构与操作，且数据操作不会改变元素状态。

访问者模式的核心是，解耦数据结构与数据操作，使得对元素的操作具备优秀的扩展性。可以通过扩展不同的数据操作类型（访问者）实现对相同元素集的不同的操作。

访问者模式的应用场景

访问者模式在生活场景中也是非常当多的，例如每年年底的 KPI 考核，KPI 考核标准是相对稳定的，但是参与 KPI 考核的员工可能每年都会发生变化，那么员工就是访问者。我们平时去食堂或者餐厅吃饭，餐厅的菜单和就餐方式是相对稳定的，但是去餐厅就餐的人员是每天都在发生变化的，因此就餐人员就是访问者。

参与 KPI 考核的人员


餐厅就餐人员

当系统中存在类型数目稳定（固定）的一类数据结构时，可以通过访问者模式方便地实现对该类型所有数据结构的不同操作，而又不会数据产生任何副作用（脏数据）。

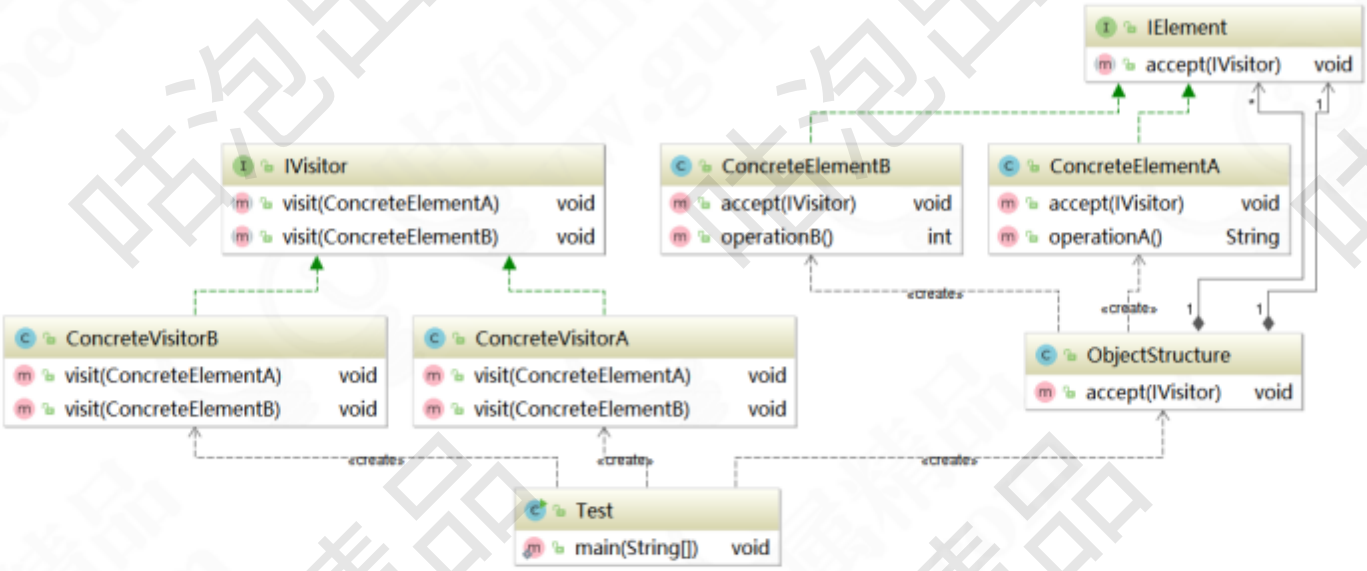
简言之，就是对集合中的不同类型数据（类型数量稳定）进行多种操作，则使用访问者模式。下面总结一下访问者模式的适用场景：

1、数据结构稳定，作用于数据结构的操作经常变化的场景；

2、需要数据结构与数据操作分离的场景；

3、需要对不同数据类型（元素）进行操作，而不使用分支判断具体类型的场景。

首先来看下访问者模式的通用 UML 类图：



从 UML 类图中，我们可以看到，访问者模式主要包含五种角色：

抽象访问者（Visitor）：接口或抽象类，该类地冠以了对每一个具体元素（Element）的访问行为 visit()方法，其参数就是具体的元素（Element）对象。理论上来说，Visitor 的方法个数与元素（Element）个数是相等的。如果元素（Element）个数经常变动，会导致 Visitor 的方法也要进行变动，此时，该情形并不适用访问者模式；

具体访问者（ConcreteVisitor）：实现对具体元素的操作；

抽象元素（Element）：接口或抽象类，定义了一个接受访问者访问的方法 accept()，表示所有元素类型都支持被访问者访问；

具体元素（Concrete Element）：具体元素类型，提供接受访问者的具体实现。通常的实现都为：visitor.visit(this)；

结构对象（ObjectStruture）：该类内部维护了元素集合，并提供方法接受访问者对该集合所有元素进行操作。

利用访问者模式实现 KPI 考核的场景

每到年底，管理层就要开始评定员工一年的工作绩效，员工分为工程师和经理；管理层有 CEO 和 CTO。那么 CTO 关注工程师的代码量、经理的新产品数量；CEO 关注的是工程师的 KPI

和经理的 KPI 以及新产品数量。

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。

```
// 员工基类
public abstract class Employee {

    public String name;
    public int kpi;// 员工 KPI

    public Employee(String name) {
        this.name = name;
        kpi = new Random().nextInt(10);
    }
    // 核心方法，接受访问者的访问
    public abstract void accept(IVisitor visitor);
}
```

Employee 类定义了员工基本信息及一个 accept()方法，accept()方法表示接受访问者的访问，由具体的子类来实现。访问者是个接口，传入不同的实现类，可访问不同的数据。下面看看工程师 Engineer 类的代码：

```
// 工程师
public class Engineer extends Employee {

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

    @Override
    public void accept(IVisitor visitor) {
        visitor.visit(this);
    }
    // 工程师一年的代码数量
    public int getCodeLines() {
        return new Random().nextInt(10 * 10000);
    }
}
```

经理 Manager 类的代码：

```
// 经理
public class Manager extends Employee {

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

    @Override
    public void accept(IVisitor visitor) {
        visitor.visit(this);
    }
    // 一年做的产品数量
    public int getProducts() {
        return new Random().nextInt(10);
```

```
    }
}
```

工程师是考核的是代码数量，经理考核的是产品数量，二者的职责不一样。也正是因为有这样的差异性，才使得访问模式能够在这个场景下发挥作用。Employee、Engineer、Manager 这 3 个类型就相当于数据结构，这些类型相对稳定，不会发生变化。

然后将这些员工添加到一个业务报表类中，公司高层可以通过该报表类的 showReport()方法查看所有员工的业绩，具体代码如下：

```
// 员工业务报表类
public class BusinessReport {

    private List<Employee> employees = new LinkedList<Employee>();

    public BusinessReport() {
        employees.add(new Manager("经理-A"));
        employees.add(new Engineer("工程师-A"));
        employees.add(new Engineer("工程师-B"));
        employees.add(new Engineer("工程师-C"));
        employees.add(new Manager("经理-B"));
        employees.add(new Engineer("工程师-D"));
    }

    /**
     * 为访问者展示报表
     * @param visitor 公司高层，如 CEO、CTO
     */
    public void showReport(IVisitor visitor) {
        for (Employee employee : employees) {
            employee.accept(visitor);
        }
    }
}
```

下面看看访问者类型的定义，访问者声明了两个 visit()方法，分别对工程师和经理访问，具体代码如下：

```
public interface IVisitor {

    // 访问工程师类型
    void visit(Engineer engineer);

    // 访问经理类型
    void visit(Manager manager);
}
```

首先定义一个 IVisitor 接口，该接口有两个 visit()方法，参数分别是 Engineer、Manager，也就是说对于 Engineer 和 Manager 的访问会调用两个不同的方法，以此达到差异化处理的目的。这两个访问者具体的实现类为 CEOVisitor 类和 CTOVisitor 类，代码如下：

```
// CEO 访问者
```

```
public class CEOVisitor implements IVisitor {

    public void visit(Engineer engineer) {
        System.out.println("工程师: " + engineer.name + ", KPI: " + engineer.kpi);
    }

    public void visit(Manager manager) {
        System.out.println("经理: " + manager.name + ", KPI: " + manager.kpi +
                ", 新产品数量: " + manager.getProducts());
    }
}
```

在 CEO 的访问者中，CEO 关注工程师的 KPI，经理的 KPI 和新产品数量，通过两个 visitor ()方法分别进行处理。如果不使用访问者模式，只通过一个 visit ()方法进行处理，那么就需要在这个 visit()方法中进行判断，然后分别处理，代码大致如下：

```
public class ReportUtil {
    public void visit(Employee employee) {
        if (employee instanceof Manager) {
            Manager manager = (Manager) employee;
            System.out.println("经理: " + manager.name + ", KPI: " + manager.kpi +
                    ", 新产品数量: " + manager.getProducts());
        } else if (employee instanceof Engineer) {
            Engineer engineer = (Engineer) employee;
            System.out.println("工程师: " + engineer.name + ", KPI: " + engineer.kpi);
        }
    }
}
```

这就导致了 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 ReportUtil 就会很复杂。而使用访问者模式，通过同一个函数对不同对元素类型进行相应对处理，使结构更加清晰、灵活性更高。

再添加一个 CTO 的访问者类：

```
public class CTOVisitor implements IVisitor {

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

    public void visit(Manager manager) {
        System.out.println("经理: " + manager.name + ", 产品数量: " + manager.getProducts());
    }
}
```

重载的 visit()方法会对元素进行不同的操作，而通过注入不同的访问者又可以替换掉访问者的具体实现，使得对元素的操作变得更灵活，可扩展性更高，同时也消除了类型转换、if-else 等“丑陋”的代码。

下面是客户端代码：

```
    public static void main(String[] args) {
      // 构建报表
      BusinessReport report = new BusinessReport();
      System.out.println("============ CEO 看报表 ============");
      report.showReport(new CEOVisitor());
      System.out.println("============ CTO 看报表 ============");
      report.showReport(new CTOVisitor());
  }
```

运行结果如下：

在上述案例中，Employee 扮演了 Element 角色，而 Engineer 和 Manager 都是 ConcreteElement；CEOVisitor 和 CTOVisitor 都是具体的 Visitor 对象；而 BusinessReport 就是 ObjectStructure。

访问者模式最大的优点就是增加访问者非常容易，我们从代码中可以看到，如果要增加一个访问者，只要新实现一个访问者接口的类，从而达到数据对象与数据操作相分离的效果。如果不实用访问者模式，而又不想对不同的元素进行不同的操作，那么必定需要使用 if-else 和类型转换，这使得代码难以升级维护。

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，使用访问者模式是否能优化我们的代码，而不是使我们的代码变得更复杂。

从静态分派到动态分派

变量被声明时的类型叫做变量的静态类型(Static Type)，有些人又把静态类型叫做明显类型(Apparent Type)；而变量所引用的对象的真实类型又叫做变量的实际类型(Actual Type)。比如：

```
List list = null;
list = new ArrayList();
```

声明了一个变量 list，它的静态类型（也叫明显类型）是 List，而它的实际类型是 ArrayList。根据对象的类型而对方法进行的选择，就是分派(Dispatch)。分派又分为两种，即静态分派和动态分派。

1、静态分派

静态分派(Static Dispatch)就是按照变量的静态类型进行分派，从而确定方法的执行版本，静态分派在编译时期就可以确定方法的版本。而静态分派最典型的应用就是方法重载,来看下面这段代码。

```
public class Main {
    public void test(String string){
        System.out.println("string");
    }

    public void test(Integer integer){
        System.out.println("integer");
    }

    public static void main(String[] args) {
        String string = "1";
        Integer integer = 1;
        Main main = new Main();
        main.test(integer);
        main.test(string);
    }
}
```

在静态分派判断的时候，我们根据多个判断依据（即参数类型和个数）判断出了方法的版本，那么这个就是多分派的概念，因为我们有一个以上的考量标准。所以 Java 是静态多分派的语言。

2、动态分派

对于动态分派，与静态相反，它不是在编译期确定的方法版本，而是在运行时才能确定。而动态分派最典型的应用就是多态的特性。举个例子，来看下面的这段代码。

```
interface Person{
    void test();
}
class Man implements Person{
    public void test(){
        System.out.println("男人");
    }
}
class Woman implements Person{
    public void test(){
        System.out.println("女人");
    }
}
public class Main {
    public static void main(String[] args) {
        Person man = new Man();
        Person woman = new Woman();
        man.test();
        woman.test();
    }
```

```
}
```

这段程序输出结果为依次打印男人和女人，然而这里的 test()方法版本，就无法根据 Man 和 Woman 的静态类型去判断了，他们的静态类型都是 Person 接口，根本无从判断。

显然，产生的输出结果，就是因为 test()方法的版本是在运行时判断的，这就是动态分派。

动态分派判断的方法是在运行时获取到 Man 和 Woman 的实际引用类型，再确定方法的版本，而由于此时判断的依据只是实际引用类型，只有一个判断依据，所以这就是单分派的概念，这时我们的考量标准只有一个，即变量的实际引用类型。相应的，这说明 Java 是动态单分派的语言。

访问者模式中的伪动态双分派

通过前面分析，我们知道 Java 是静态多分派、动态单分派的语言。Java 底层不支持动态的双分派。但是通过使用设计模式，也可以在 Java 语言里实现伪动态双分派。在访问者模式中使用的就是伪动态双分派。所谓动态双分派就是在运行时依据两个实际类型去判断一个方法的运行行为，而访问者模式实现的手段是进行了两次动态单分派来达到这个效果。

还是回到前面的 KPI 考核业务场景当中，BusinessReport 类中的 showReport()方法：

```
public void showReport(IVisitor visitor) {
    for (Employee employee : employees) {
        employee.accept(visitor);
    }
}
```

这里就是依据 Employee 和 IVisitor 两个实际类型决定了 showReport()方法的执行结果，从而决定了 accept()方法的动作。

分析 accept()方法的调用过程

1.当调用 accept()方法时 ,根据 Employee 的实际类型决定是调用 Engineer 还是 Manager 的 accept()方法。

2.这时 accept()方法的版本已经确定，假如是 Engineer，它的 accept()方法是调用下面这行代码。

```
public void accept(IVisitor visitor) {
    visitor.visit(this);
}
```

此时的 this 是 Engineer 类型，所以对应的是 IVisitor 接口的 visit(Engineer engineer)方法，此时需要再根据访问者的实际类型确定 visit()方法的版本，如此一来，就完成了动态双分派的过程。

以上的过程就是通过两次动态双分派，第一次对 accept()方法进行动态分派，第二次对访问者的 visit()方法进行动态分派，从而达到了根据两个实际类型确定一个方法的行为的效果。

而原本我们的做法，通常是传入一个接口，直接使用该接口的方法，此为动态单分派，就像策略模式一样。在这里，showReport()方法传入的访问者接口并不是直接调用自己的 visit()方法，而是通过 Employee 的实际类型先动态分派一次，然后在分派后确定的方法版本里再进行自己的动态分派。

注意：这里确定 accept(IVisitor visitor)方法是静态分派决定的，所以这个并不在此次动态双分派的范畴内，而且静态分派是在编译期就完成的，所以 accept(IVisitor visitor)方法的静态分派与访问者模式的动态双分派并没有任何关系。动态双分派说到底还是动态分派，是在运行时发生的，它与静态分派有着本质上的区别，不可以说一次动态分派加一次静态分派就是动态双分派，而且访问者模式的双分派本身也是另有所指。

而 this 的类型不是动态确定的，你写在哪个类当中，它的静态类型就是哪个类，这是在编译期就确定的，不确定的是它的实际类型，请小伙伴也要区分开来。

访问者模式在源码中的应用

首先来看 JDK 的 NIO 模块下的 FileVisitor,它接口提供了递归遍历文件树的支持。这个接口上的方法表示了遍历过程中的关键过程，允许你在文件被访问、目录将被访问、目录已被访问、发生错误等等过程上进行控制；换句话说，这个接口在文件被访问前、访问中和访问后，以及产生错误的时候都有相应的钩子程序进行处理。

调用 FileVisitor 中的方法，会返回访问结果 FileVisitResult 对象值，用于决定当前操作完成后接下来该如何处理。FileVisitResult 的标准返回值存放到 FileVisitResult 枚举类型中：

FileVisitResult.CONTINUE：这个访问结果表示当前的遍历过程将会继续。

FileVisitResult.SKIP_SIBLINGS：这个访问结果表示当前的遍历过程将会继续，但是要忽略当前文件/目录的兄弟节点。

FileVisitResult.SKIP_SUBTREE：这个访问结果表示当前的遍历过程将会继续，但是要忽略当前目录下的所有节点。

FileVisitResult.TERMINATE：这个访问结果表示当前的遍历过程将会停止。

```
public interface FileVisitor<T> {

    FileVisitResult preVisitDirectory(T dir, BasicFileAttributes attrs)
        throws IOException;

    FileVisitResult visitFile(T file, BasicFileAttributes attrs)
        throws IOException;

    FileVisitResult visitFileFailed(T file, IOException exc)
        throws IOException;

    FileVisitResult postVisitDirectory(T dir, IOException exc)
        throws IOException;
}
```

通过它去遍历文件树会比较方便，比如查找文件夹内符合某个条件的文件或者某一天内所创建的文件，这个类中都提供了相对应的方法。我们来看一下它的实现其实也非常简单：

```
public class SimpleFileVisitor<T> implements FileVisitor<T> {
    protected SimpleFileVisitor() {
    }

    @Override
    public FileVisitResult preVisitDirectory(T dir, BasicFileAttributes attrs)
        throws IOException
    {
        Objects.requireNonNull(dir);
        Objects.requireNonNull(attrs);
        return FileVisitResult.CONTINUE;
    }

    @Override
    public FileVisitResult visitFile(T file, BasicFileAttributes attrs)
        throws IOException
    {
        Objects.requireNonNull(file);
        Objects.requireNonNull(attrs);
        return FileVisitResult.CONTINUE;
    }

    @Override
    public FileVisitResult visitFileFailed(T file, IOException exc)
        throws IOException
    {
```

```
        Objects.requireNonNull(file);
        throw exc;
    }

    @Override
    public FileVisitResult postVisitDirectory(T dir, IOException exc)
        throws IOException
    {
        Objects.requireNonNull(dir);
        if (exc != null)
            throw exc;
        return FileVisitResult.CONTINUE;
    }
}
```

下面再来看访问者模式在 Spring 中的应用，Spring IoC 中有个 BeanDefinitionVisitor 类，其中有一个 visitBeanDefinition()方法，我们来看他的源码：

```
public class BeanDefinitionVisitor {

    @Nullable
    private StringValueResolver valueResolver;


    public BeanDefinitionVisitor(StringValueResolver valueResolver) {
        Assert.notNull(valueResolver, "StringValueResolver must not be null");
        this.valueResolver = valueResolver;
    }

    protected BeanDefinitionVisitor() {
    }

    public void visitBeanDefinition(BeanDefinition beanDefinition) {
        visitParentName(beanDefinition);
        visitBeanClassName(beanDefinition);
        visitFactoryBeanName(beanDefinition);
        visitFactoryMethodName(beanDefinition);
        visitScope(beanDefinition);
        if (beanDefinition.hasPropertyValues()) {
            visitPropertyValues(beanDefinition.getPropertyValues());
        }
        if (beanDefinition.hasConstructorArgumentValues()) {
            ConstructorArgumentValues cas = beanDefinition.getConstructorArgumentValues();
            visitIndexedArgumentValues(cas.getIndexedArgumentValues());
            visitGenericArgumentValues(cas.getGenericArgumentValues());
        }
    }
     ...
}
```

我们看到在 visitBeanDefinition()方法中，分别访问了其他的数据，比如父类的名字、自己的类名、在 IoC 容器中的名称等各种信息。

访问者模式的优缺点

优点：

1、解耦了数据结构与数据操作，使得操作集合可以独立变化；

2、扩展性好：可以通过扩展访问者角色，实现对数据集的不同操作；

3、元素具体类型并非单一，访问者均可操作；

4、各角色职责分离，符合单一职责原则。

缺点：

1、无法增加元素类型：若系统数据结构对象易于变化，经常有新的数据对象增加进来，则访问者类必须增加对应元素类型的操作，违背了开闭原则；

2、具体元素变更困难：具体元素增加属性，删除属性等操作会导致对应的访问者类需要进行相应的修改，尤其当有大量访问者类时，修改范围太大；

3、违背依赖倒置原则：为了达到”区别对待“，访问者依赖的是具体元素类型，而不是抽象。