

69-访问者模式（下）：为什么支持双分派的语言不需要访问者模式？

上一节课中，我们学习了访问者模式的原理和实现，并且还原了访问者模式诞生的思维过程。总体上来讲，这个模式的代码实现比较难，所以应用场景并不多。从应用开发的角度来说，它的确不是我们学习的重点。

不过，我们前面反复说过，学习我的专栏，并不只是让你掌握知识，更重要的是锻炼你分析、解决问题的能力，锻炼你的逻辑思维能力，所以，今天我们继续把访问者模式作为引子，一块讨论一下这样两个问题，希望能激发你的深度思考：

- 为什么支持双分派的语言不需要访问者模式呢？
- 除了访问者模式，上一节课中的例子还有其他实现方案吗？

话不多说，让我们正式开始今天的学习吧！

为什么支持双分派的语言不需要访问者模式？

实际上，讲到访问者模式，大部分书籍或者资料都会讲到Double Dispatch，中文翻译为双分派。虽然学习访问者模式，并不用非得理解这个概念，我们前面的讲解就没有提到它，但是，为了让你在查看其它书籍或者资料的时候，不会卡在这个概念上，我觉得有必要在这里讲一下。

除此之外，我觉得，学习Double Dispatch还能加深你对访问者模式的理解，而且能一并帮你搞清楚今天文章标题中的这个问题：为什么支持双分派的语言就不需要访问者模式？这个问题在面试中可是会被问到的哦！

既然有Double Dispatch，对应的就有Single Dispatch。所谓**Single Dispatch**，指的是执行哪个对象的方法，根据对象的运行时类型来决定；执行对象的哪个方法，根据方法参数的编译时类型来决定。所谓**Double Dispatch**，指的是执行哪个对象的方法，根据对象的运行时类型来决定；执行对象的哪个方法，根据方法参数的运行时类型来决定。

如何理解“Dispatch”这个单词呢？ 在面向对象编程语言中，我们可以把方法调用理解为一种消息传递，也就是“Dispatch”。一个对象调用另一个对象的方法，就相当于给它发送一条消息。这条消息起码要包含对象名、方法名、方法参数。

如何理解“Single”“Double”这两个单词呢？ “Single”“Double”指的是执行哪个对象的哪个方法，跟几个因素的运行时类型有关。我们进一步解释一下。Single Dispatch之所以称为“Single”，是因为执行哪个对象的哪个方法，只跟“对象”的运行时类型有关。Double Dispatch之所以称为“Double”，是因为执行哪个对象的哪个方法，跟“对象”和“方法参数”两者的运行时类型有关。

具体到编程语言的语法机制，Single Dispatch和Double Dispatch跟多态和函数重载直接相关。当前主流的面向对象编程语言（比如，Java、C++、C#）都只支持Single Dispatch，不支持Double Dispatch。

接下来，我们拿Java语言来举例说明一下。

Java支持多态特性，代码可以在运行时获得对象的实际类型（也就是前面提到的运行时类型），然后根据实际类型决定调用哪个方法。尽管Java支持函数重载，但Java设计的函数重载的语法规则是，并不是在运行时，根据传递进函数的参数的实际类型，来决定调用哪个重载函数，而是在编译时，根据传递进函数的参数的声明类型（也就是前面提到的编译时类型），来决定调用哪个重载函数。也就是说，具体执行哪个对象

的哪个方法，只跟对象的运行时类型有关，跟参数的运行时类型无关。所以，Java语言只支持Single Dispatch。

这么说比较抽象，我举个例子来具体说明一下，代码如下所示：

```
public class ParentClass {
    public void f() {
        System.out.println("I am ParentClass's f().");
    }
}

public class ChildClass extends ParentClass {
    public void f() {
        System.out.println("I am ChildClass's f().");
    }
}

public class SingleDispatchClass {
    public void polymorphismFunction(ParentClass p) {
        p.f();
    }

    public void overloadFunction(ParentClass p) {
        System.out.println("I am overloadFunction(ParentClass p).");
    }

    public void overloadFunction(ChildClass c) {
        System.out.println("I am overloadFunction(ChildClass c).");
    }
}

public class DemoMain {
    public static void main(String[] args) {
        SingleDispatchClass demo = new SingleDispatchClass();
        ParentClass p = new ChildClass();
        demo.polymorphismFunction(p); // 执行哪个对象的方法，由对象的实际类型决定
        demo.overloadFunction(p); // 执行对象的哪个方法，由参数对象的声明类型决定
    }
}

// 代码执行结果：
I am ChildClass's f().
I am overloadFunction(ParentClass p).
```

在上面的代码中，第31行代码的polymorphismFunction()函数，执行p的实际类型的f()函数，也就是ChildClass的f()函数。第32行代码的overloadFunction()函数，匹配的是重载函数中的overloadFunction(ParentClass p)，也就是根据p的声明类型来决定匹配哪个重载函数。

假设Java语言支持Double Dispatch，那下面的代码（摘抄自上节课中第二段代码，建议结合上节课的讲解一块理解）中的第37行就不会报错。代码会在运行时，根据参数（resourceFile）的实际类型（PdfFile、PPTFile、WordFile），来决定使用extract2txt的三个重载函数中的哪一个。那下面的代码实现就能正常运行了，也就不需要访问者模式了。这也回答了为什么支持Double Dispatch的语言不需要访问者模式。

```

public abstract class ResourceFile {
    protected String filePath;
    public ResourceFile(String filePath) {
        this.filePath = filePath;
    }
}

public class PdfFile extends ResourceFile {
    public PdfFile(String filePath) {
        super(filePath);
    }
    //...
}
//...PPTFile、WordFile代码省略...
public class Extractor {
    public void extract2txt(PPTFile pptFile) {
        //...
        System.out.println("Extract PPT.");
    }

    public void extract2txt(PdfFile pdfFile) {
        //...
        System.out.println("Extract PDF.");
    }

    public void extract2txt(WordFile wordFile) {
        //...
        System.out.println("Extract WORD.");
    }
}

public class ToolApplication {
    public static void main(String[] args) {
        Extractor extractor = new Extractor();
        List<ResourceFile> resourceFiles = listAllResourceFiles(args[0]);
        for (ResourceFile resourceFile : resourceFiles) {
            extractor.extract2txt(resourceFile);
        }
    }

    private static List<ResourceFile> listAllResourceFiles(String resourceDirectory) {
        List<ResourceFile> resourceFiles = new ArrayList<>();
        //...根据后缀(pdf/ppt/word)由工厂方法创建不同的类对象(PdfFile/PPTFile/WordFile)
        resourceFiles.add(new PdfFile("a.pdf"));
        resourceFiles.add(new WordFile("b.word"));
        resourceFiles.add(new PPTFile("c.ppt"));
        return resourceFiles;
    }
}

```

除了访问者模式，上一节的例子还有其他实现方案吗？

上节课，我通过一个例子来给你展示了，访问者模式是如何一步一步设计出来的。我们这里再一块回顾一下那个例子。我们从网站上爬取了很多资源文件，它们的格式有三种：PDF、PPT、Word。我们要开发一个工具来处理这批资源文件，这其中就包含抽取文本内容、压缩资源文件、提取文件元信息等。

实际上，开发这个工具有很多种代码设计和实现思路。为了讲解访问者模式，上节课我们选择了用访问者模式来实现。实际上，我们还有其他的实现方法，比如，我们还可以利用工厂模式来实现，定义一个包含

extract2txt()接口函数的Extractor接口。PdfExtractor、PPTExtractor、WordExtractor类实现Extractor接口，并且在各自的extract2txt()函数中，分别实现Pdf、PPT、Word格式文件的文本内容抽取。ExtractorFactory工厂类根据不同的文件类型，返回不同的Extractor。

这个实现思路其实更加简单，我们直接看代码。

```
public abstract class ResourceFile {
    protected String filePath;
    public ResourceFile(String filePath) {
        this.filePath = filePath;
    }
    public abstract ResourceFileType getType();
}

public class PdfFile extends ResourceFile {
    public PdfFile(String filePath) {
        super(filePath);
    }

    @Override
    public ResourceFileType getType() {
        return ResourceFileType.PDF;
    }

    //...
}

//...PPTFile/WordFile跟PdfFile代码结构类似，此处省略...

public interface Extractor {
    void extract2txt(ResourceFile resourceFile);
}

public class PdfExtractor implements Extractor {
    @Override
    public void extract2txt(ResourceFile resourceFile) {
        //...
    }
}

//...PPTExtractor/WordExtractor跟PdfExtractor代码结构类似，此处省略...

public class ExtractorFactory {
    private static final Map<ResourceFileType, Extractor> extractors = new HashMap<>();
    static {
        extractors.put(ResourceFileType.PDF, new PdfExtractor());
        extractors.put(ResourceFileType.PPT, new PPTExtractor());
        extractors.put(ResourceFileType.WORD, new WordExtractor());
    }

    public static Extractor getExtractor(ResourceFileType type) {
        return extractors.get(type);
    }
}

public class ToolApplication {
    public static void main(String[] args) {
        List<ResourceFile> resourceFiles = listAllResourceFiles(args[0]);
        for (ResourceFile resourceFile : resourceFiles) {
            Extractor extractor = ExtractorFactory.getExtractor(resourceFile.getType());
            extractor.extract2txt(resourceFile);
        }
    }
}
```

```
    }  
}  
  
private static List<ResourceFile> listAllResourceFiles(String resourceDirectory) {  
    List<ResourceFile> resourceFiles = new ArrayList<>();  
    //...根据后缀(pdf/ppt/word)由工厂方法创建不同的类对象(PdfFile/PPTFile/WordFile)  
    resourceFiles.add(new PdfFile("a.pdf"));  
    resourceFiles.add(new WordFile("b.word"));  
    resourceFiles.add(new PPTFile("c.ppt"));  
    return resourceFiles;  
}  
}
```

当需要添加新的功能的时候，比如压缩资源文件，类似抽取文本内容功能的代码实现，我们只需要添加一个Compressor接口，PdfCompressor、PPTCompressor、WordCompressor三个实现类，以及创建它们的CompressorFactory工厂类即可。唯一需要修改的只有最上层的ToolApplication类。基本上符合“对扩展开放、对修改关闭”的设计原则。

对于资源文件处理工具这个例子，如果工具提供的功能并不是非常多，只有几个而已，那我更推荐使用工厂模式的实现方式，毕竟代码更加清晰、易懂。相反，如果工具提供非常多的功能，比如有十几个，那我更推荐使用访问者模式，因为访问者模式需要定义的类型要比工厂模式的实现方式少很多，类太多也会影响到代码的可维护性。

重点回顾

好了，今天内容到此就讲完了。我们一块来总结回顾一下，你需要重点掌握的内容。

总体上来讲，访问者模式难以理解，应用场景有限，不是特别必需，我不建议在项目中使用它。所以，对于上节课中的处理资源文件的例子，我更推荐使用工厂模式来设计和实现。

除此之外，我们今天重点讲解了Double Dispatch。在面向对象编程语言中，方法调用可以理解作为一种消息传递（Dispatch）。一个对象调用另一个对象的方法，就相当于给它发送一条消息，这条消息起码要包含对象名、方法名和方法参数。

所谓Single Dispatch，指的是执行哪个对象的方法，根据对象的运行时类型来决定；执行对象的哪个方法，根据方法参数的编译时类型来决定。所谓Double Dispatch，指的是执行哪个对象的方法，根据对象的运行时类型来决定；执行对象的哪个方法，根据方法参数的运行时类型来决定。

具体到编程语言的语法机制，Single Dispatch和Double Dispatch跟多态和函数重载直接相关。当前主流的面向对象编程语言（比如，Java、C++、C#）都只支持Single Dispatch，不支持Double Dispatch。

课堂讨论

1. 访问者模式将操作与对象分离，是否违背面向对象设计原则？你怎么看待这个问题呢？
2. 在解释Single Dispatch的代码示例中，如果我们把SingleDispatchClass的代码改成下面这样，其他代码不变，那DemoMain的输出结果会是什么呢？为什么会是这样的结果呢？

```
public class SingleDispatchClass {
```

```
public void polymorphismFunction(ParentClass p) {  
    p.f();  
}  
  
public void overloadFunction(ParentClass p) {  
    p.f();  
}  
  
public void overloadFunction(ChildClass c) {  
    c.f();  
}  
}
```

欢迎留言和我分享你的想法。如果有收获，也欢迎你把这篇文章分享给你的朋友。

精选留言：

- 漫游者 2020-04-10 08:52:03
我觉得应该不违背面相对象设计原则。原因是对象包含的属性和方法应该以对象履行的职责为界限。访问者模式里的操作应该算是一个项新的职能，而不是对象里的一个方法。