

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？

王争 · 设计模式之美



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：

 复制代码

```
1 public class ParentClass {
2     public void f() {
3         System.out.println("I am ParentClass's f().");
4     }
5 }
6
7 public class ChildClass extends ParentClass {
8     public void f() {
9         System.out.println("I am ChildClass's f().");
10    }
11 }
12
13 public class SingleDispatchClass {
14     public void polymorphismFunction(ParentClass p) {
15         p.f();
16     }
17
18     public void overloadFunction(ParentClass p) {
19         System.out.println("I am overloadFunction(ParentClass p).");
20     }
21
22     public void overloadFunction(ChildClass c) {
23         System.out.println("I am overloadFunction(ChildClass c).");
24     }
25 }
26
27 public class DemoMain {
28     public static void main(String[] args) {
29         SingleDispatchClass demo = new SingleDispatchClass();
30         ParentClass p = new ChildClass();
31         demo.polymorphismFunction(p); // 执行哪个对象的方法，由对象的实际类型决定
32         demo.overloadFunction(p); // 执行对象的哪个方法，由参数对象的声明类型决定
33     }
34 }
35
36 // 代码执行结果：
37 I am ChildClass's f().
```

```
38 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。

 复制代码

```
1 public abstract class ResourceFile {
2     protected String filePath;
3     public ResourceFile(String filePath) {
4         this.filePath = filePath;
5     }
6 }
7
8 public class PdfFile extends ResourceFile {
9     public PdfFile(String filePath) {
10         super(filePath);
11     }
12     //...
13 }
14 //...PPTFile、WordFile代码省略...
15 public class Extractor {
16     public void extract2txt(PPTFile pptFile) {
17         //...
18         System.out.println("Extract PPT.");
19     }
20
21     public void extract2txt(PdfFile pdfFile) {
22         //...
23         System.out.println("Extract PDF.");
24     }
25
26     public void extract2txt(WordFile wordFile) {
```

```

27     //...
28     System.out.println("Extract WORD.");
29 }
30 }
31
32 public class ToolApplication {
33     public static void main(String[] args) {
34         Extractor extractor = new Extractor();
35         List<ResourceFile> resourceFiles = listAllResourceFiles(args[0]);
36         for (ResourceFile resourceFile : resourceFiles) {
37             extractor.extract2txt(resourceFile);
38         }
39     }
40
41     private static List<ResourceFile> listAllResourceFiles(String resourceDirectory) {
42         List<ResourceFile> resourceFiles = new ArrayList<>();
43         //...根据后缀(pdf/ppt/word)由工厂方法创建不同的类对象(PdfFile/PPTFile/WordFile)
44         resourceFiles.add(new PdfFile("a.pdf"));
45         resourceFiles.add(new WordFile("b.word"));
46         resourceFiles.add(new PPTFile("c.ppt"));
47         return resourceFiles;
48     }
49 }

```

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。

```

1 public abstract class ResourceFile {
2     protected String filePath;
3     public ResourceFile(String filePath) {
4         this.filePath = filePath;
5     }
6     public abstract ResourceType getType();
7 }
8
9 public class PdfFile extends ResourceFile {
10     public PdfFile(String filePath) {
11         super(filePath);
12     }
13
14     @Override
15     public ResourceType getType() {
16         return ResourceType.PDF;
17     }
18
19     //...
20 }
21
22 //...PPTFile/WordFile跟PdfFile代码结构类似，此处省略...
23
24 public interface Extractor {
25     void extract2txt(ResourceFile resourceFile);
26 }
27
28 public class PdfExtractor implements Extractor {
29     @Override
30     public void extract2txt(ResourceFile resourceFile) {
31         //...
32     }
33 }
34
35 //...PPTExtractor/WordExtractor跟PdfExtractor代码结构类似，此处省略...
36
37 public class ExtractorFactory {
38     private static final Map<ResourceFileType, Extractor> extractors = new HashMap<
39     static {
40         extractors.put(ResourceFileType.PDF, new PdfExtractor());
41         extractors.put(ResourceFileType.PPT, new PPTExtractor());
42         extractors.put(ResourceFileType.WORD, new WordExtractor());
43     }
44
45     public static Extractor getExtractor(ResourceFileType type) {
46         return extractors.get(type);
47     }

```

```
48 }
49
50 public class ToolApplication {
51     public static void main(String[] args) {
52         List<ResourceFile> resourceFiles = listAllResourceFiles(args[0]);
53         for (ResourceFile resourceFile : resourceFiles) {
54             Extractor extractor = ExtractorFactory.getExtractor(resourceFile.getType())
55             extractor.extract2txt(resourceFile);
56         }
57     }
58
59     private static List<ResourceFile> listAllResourceFiles(String resourceDirectory
60         List<ResourceFile> resourceFiles = new ArrayList<>();
61         //...根据后缀(pdf/ppt/word)由工厂方法创建不同的类对象(PdfFile/PPTFile/WordFile)
62         resourceFiles.add(new PdfFile("a.pdf"));
63         resourceFiles.add(new WordFile("b.word"));
64         resourceFiles.add(new PPTFile("c.ppt"));
65         return resourceFiles;
66     }
67 }
```

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？

 复制代码

```
1 public class SingleDispatchClass {
2     public void polymorphismFunction(ParentClass p) {
3         p.f();
4     }
5
6     public void overloadFunction(ParentClass p) {
7         p.f();
8     }
9
10    public void overloadFunction(ChildClass c) {
11        c.f();
12    }
13 }
```

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。

本文讨论了支持双分派的语言为何不需要访问者模式。首先介绍了Single Dispatch和Double Dispatch的概念，并以Java语言为例说明了Java只支持Single Dispatch的特性。接着通过代码示例展示了如果Java语言支持Double Dispatch，就不需要访问者模式的情况。文章还提到了其他实现方案，如工厂模式，来处理资源文件，以及访问者模式和双分派的应用。总体上，文章强调了访问者模式难以理解，应用场景有限，不建议在项目中使用。此外，重点讲解了Double Dispatch的概念，以及在面向对象编程语言中的应用。文章通过清晰的概念解释和具体的代码示例，帮助读者理解了访问者模式和双分派的概念，以及它们在面向对象编程语言中的应用。

© 版权归极客邦科技所有，未经许可不得传播售卖。页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

全部留言 (51)

最新 精选



DFighting

2020-11-22

在面向对象的世界里，万物都可以作为对象，操作如果不需要扩展，那么应该聚合在对象之中，但是如果操作需要扩展，那么这个就可以单独拿出来，代码设计中的面向对象不是一定需要和现实世界的理解的对象一致的，程序未来的演进方向在何处，面向对象就该如何抽象；至于问题，被调用对象的这一个多态在Single Dispatch中也是支持的

作者回复: 嗯嗯 🤔🤔🤔🤔🤔



👍 8



Laughing

2020-11-26

1. 个人认为并不违反面向对象的设计原则，基于单分派语言的特性，访问者模式本质上是解决了没有指定对象实际类型时函数重载的问题，这是语言特性导致的，开闭原则指的是“对修改关闭，对扩展开放”，理论上该模式并没有实质上的修改操作，更多的是通过另一种形式完成了功能扩展。
2. I am ChildClass's f() 输出两次，因为运行时对象的实际类型是child，所以会执行child的f函数

作者回复: 嗯嗯 🤔🤔🤔🤔🤔



👍 2



大悟

2020-08-30

感觉访问者模式的核心不在于双重分派，而是在于将数据对象和操作分离的思想。双重分派只是由于 Java 只支持单分派而采取的手段，不是目的。上文中说理解双重分派是理解访问者模式的核心，实在不太认同，感觉有点舍本逐末了。

作者回复: 所有的行为模式都是将行为分离，这点不是访问者模式独有的。如果支持double dispatch，完全没必要有访问者模式。



👍 1



小晏子

2020-04-10

课后思考：

1. 看要怎么理解这个问题了，简单来看将操作与对象分离是违背了面向对象的设计原则，但从另外的角度看，将操作也看做对象，然后将不同的对象进行组合，那么并不违背面向对象的设计，而且在访问者模式中，操作可能不断增加，是属于是变化比较多的，将不变的部分和可变的分开，然后使用组合的方式使用是符合面向对象设计的。

2. 会输出：

I am ChildClass's f().

I am ChildClass's f().

调用demo.overloadFunction(p);时，会根据重载特性调用函数

```
public void overloadFunction(ParentClass p)
```

```
{
```

```
    p.f();
```

```
}
```

运行时，因为p是ChildClass对象，所以会根据多态特性使用ChildClass的f函数。

共 3 条评论 >

👍 104



,

2020-04-10

关于访问者模式的替代方式,我的看法:

先放总结: 行为不可抽象+水平扩展较多-->工厂模式更合适

行为可抽象+垂直扩展较多-->模板方法模式更合适

我认为模板方法模式和工厂模式都可以,具体使用哪种,应该根据扩展的方向来确认:
当前的场景是对不同文件格式的文本进行处理,目前有word,ppt与pdf三种格式,他们的行为都

不一致,比如word的抽取,分析与pdf的抽取,分析行为不一致,而且扩展的方向是添加不同的文件格式,比如txt格式,excel等格式,那么最好的方式就是采用工厂模式,每次添加新格式需要添加新的工厂,实现相应的方法

如果扩展的方向是给不同文件格式添加更多的功能,同时这些行为可以抽象出来,比如当前有抽取,分析,压缩等功能,他们有很大一部分可以抽象到父类,那么我要给所有的文件格式添加敏感词替换,格式化文本等功能,就可以将它们添加到父类,而不用每个工厂都加一遍,这种情况模板方法模式更合适

共 6 条评论 >

👍 31



迷羊

2020-04-10

1.争哥在前面讲面向对象的设计原则时就已经解答了这个问题,不要太死板的遵守各种设置原则,定义,只要写出来的代码是可扩展、可读性、可复用的代码就是好代码。

2.代码执行结果

I am ChildClass's f().

I am ChildClass's f().

虽然执行重载方法时是根据参数的编译时类型,但是调用哪个对象的方法是根据对象的运行时类型来决定的,所以最终调用的还是实际类型的f()方法。



👍 27



寒光

2022-06-08

老师举的例子不太好,访问者模式更多是用来在一组对象中收集信息,然后汇总。

由于不同对象提供的信息不一样,所以才会有多分派,因而访问者会提供不同的访问方法,这些方法应该有更可读的名字,而不应该是统一的visit重载方法。



👍 5



zj

2020-04-13

实际上操作与对象并没有分开吧,访问者accept方法其实就是操作了,只不过将操作部分抽象出来了,组合到对象里而已

共 2 条评论 >

👍 5



Frank

2020-04-11

打卡 今日学习访问者模式下，收获如下：

访问者模式实现比较难于理解，主要要理解静态分派和动态分派。通过本专栏的内容学习到了双分派和单分派。自己使用的主要语言Java是单分派。单分派就是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。

理解分派之前需要理解变量是有静态类型和实际类型的，如 `A a = new B()`，变量a的静态类型（声明类型）是A，实际类型是B。如果是 `A a = new A()`，那么变量a的静态类型和实际类型都是A。方法调用过程中判断是用父类对象还是子类对象其实就是多态，运行时根据变量的实际类型来决定是使用子类对象中的方法还是父类对象中的方法。其中涉及到invokevirtual字节码多态查找流程，简单的理解就是先在已确定对象中寻找方法（如子类），如果找不到往父类中找，如果一直找不到就抛出异常。在确定调用对象后（如确定是子类对象）在调用方法时可能存在方法的重载，这时候就涉及到静态分派（静态绑定），根据变量的静态类型（声明类型）来判断方法的调用版本。

课后思考：

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？对于这个问题，我觉得不能死套设计原则，对于业务场景，要有所取舍。就像专栏中的这个例子，如果所有的功能都写在了相关类中，随着需求不断的迭代，类会变膨胀，可维护性、可读性、可测试性都会变差，后期维护成本会变高。如果一开始就能确定需求不会变化，就只有这么两类操作，那么可以不用访问者这么模式，直接写在相关类中。

2. DemoMain 的输出结果会是什么呢？ChildClass 类中的f()方法会被调用两次。首先根据方法重载是静态绑定，会调用形参是ParentClass的overloadFunction方法，在该方法中“p.f();”变量“p”的静态类型是

ParentClass，而实际类型却是ChildClass。根据多态的动态绑定，在ChildClass类中复写了父类中的f()方法，因此，这里会调用ChildClass中的f()；



4



88591

2020-04-14

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？我理解是没有违背，面向对象里面的封装应该是针对 属性与操作都是比较明确的情况下。访问者模式模式中的操作实际是不确定，不稳定的。所以将这部分与对象分离出来了。也就是分离了对象中稳定的（抽象出共性）与非稳定性的部分。因为我们不确定这个对象后续还要添加什么操作，那么我们就定义一个操作，可以操作对象的数据（面向对象中的抽象）。



3