

## 68 | 访问者模式（上）：手把手带你还原访问者模式诞生的思维过程

王争 · 设计模式之美



前面我们讲到，大部分设计模式的原理和实现都很简单，不过也有例外，比如今天要讲的访问者模式。它可以算是 23 种经典设计模式中最难理解的几个之一。因为它难理解、难实现，应用它会导致代码的可读性、可维护性变差，所以，访问者模式在实际的软件开发中很少被用到，在没有特别必要的情况下，建议你不要使用访问者模式。


尽管如此，为了让你以后读到应用了访问者模式的代码的时候，能一眼就能看出代码的设计意图，同时为了整个专栏内容的完整性，我觉得还是有必要给你讲一讲这个模式。除此之外，为了最大化学习效果，我今天不只是单纯地讲解原理和实现，更重要的是，我会手把手带你还原访问者模式诞生的思维过程，让你切身感受到创造一种新的设计模式出来并不是件难事。

话不多说，让我们正式开始今天的学习吧！

**带你“发明”访问者模式**

假设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。我们现在要开发一个工具来处理这批资源文件。这个工具的其中一个功能是，把这些资源文件中的文本内容抽取出来放到 txt 文件中。如果让你来实现，你会怎么做呢？

实现这个功能并不难，不同的人有不同的写法，我将其中一种代码实现方式贴在这里。其中，ResourceFile 是一个抽象类，包含一个抽象函数 extract2txt()。PdfFile、PPTFile、WordFile 都继承 ResourceFile 类，并且重写了 extract2txt() 函数。在 ToolApplication 中，我们可以利用多态特性，根据对象的实际类型，来决定执行哪个方法。

 复制代码

```
1 public abstract class ResourceFile {
2     protected String filePath;
3
4     public ResourceFile(String filePath) {
5         this.filePath = filePath;
6     }
7
8     public abstract void extract2txt();
9 }
10
11 public class PPTFile extends ResourceFile {
12     public PPTFile(String filePath) {
13         super(filePath);
14     }
15
16     @Override
17     public void extract2txt() {
18         //...省略一大坨从PPT中抽取文本的代码...
19         //...将抽取出来的文本保存在跟filePath同名的.txt文件中...
20         System.out.println("Extract PPT.");
21     }
22 }
23
24 public class PdfFile extends ResourceFile {
25     public PdfFile(String filePath) {
26         super(filePath);
27     }
28
29     @Override
30     public void extract2txt() {
31         //...
32         System.out.println("Extract PDF.");
33     }
34 }
```

```

35
36 public class WordFile extends ResourceFile {
37     public WordFile(String filePath) {
38         super(filePath);
39     }
40
41     @Override
42     public void extract2txt() {
43         //...
44         System.out.println("Extract WORD.");
45     }
46 }
47
48 // 运行结果是:
49 // Extract PDF.
50 // Extract WORD.
51 // Extract PPT.
52 public class ToolApplication {
53     public static void main(String[] args) {
54         List<ResourceFile> resourceFiles = listAllResourceFiles(args[0]);
55         for (ResourceFile resourceFile : resourceFiles) {
56             resourceFile.extract2txt();
57         }
58     }
59
60     private static List<ResourceFile> listAllResourceFiles(String resourceDirectory)
61     List<ResourceFile> resourceFiles = new ArrayList<>();
62     //...根据后缀(pdf/ppt/word)由工厂方法创建不同的类对象(PdfFile/PPTFile/WordFile)
63     resourceFiles.add(new PdfFile("a.pdf"));
64     resourceFiles.add(new WordFile("b.word"));
65     resourceFiles.add(new PPTFile("c.ppt"));
66     return resourceFiles;
67 }
68 }

```


如果工具的功能不停地扩展，不仅要能抽取文本内容，还要支持压缩、提取文件元信息（文件名、大小、更新时间等等）构建索引等一系列的功能，那如果我们继续按照上面的实现思路，就会存在这样几个问题：

违背开闭原则，添加一个新的功能，所有类的代码都要修改；

虽然功能增多，每个类的代码都不断膨胀，可读性和可维护性都变差了；

把所有比较上层的业务逻辑都耦合到 PdfFile、PPTFile、WordFile 类中，导致这些类的职责不够单一，变成了大杂烩。

针对上面的问题，我们常用的解决方法就是拆分解耦，把业务操作跟具体的数据结构解耦，设计成独立的类。这里我们按照访问者模式的演进思路来对上面的代码进行重构。重构之后的代码如下所示。

 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 }
```


这其中最关键的一点设计是，我们把抽取文本内容的操作，设计成了三个重载函数。函数重载是 Java、C++ 这类面向对象编程语言中常见的语法机制。所谓重载函数是指，在同一类中函数名相同、参数不同的一组函数。

不过，如果你足够细心，就会发现，上面的代码是编译通过不了的，第 37 行会报错。这是为什么呢？

我们知道，多态是一种动态绑定，可以在运行时获取对象的实际类型，来运行实际类型对应的方法。而函数重载是一种静态绑定，在编译时并不能获取对象的实际类型，而是根据声明类型执行声明类型对应的方法。

在上面代码的第 35~38 行中，resourceFiles 包含的对象的声明类型都是 ResourceFile，而我们并没有在 Extractor 类中定义参数类型是 ResourceFile 的 extract2txt() 重载函数，所以在编译阶段就通过不了，更别说在运行时根据对象的实际类型执行不同的重载函数了。那如何解决这个问题呢？

解决的办法稍微有点难理解，我们先来看代码，然后我再来给你慢慢解释。

 复制代码

```
1 public abstract class ResourceFile {
2     protected String filePath;
3     public ResourceFile(String filePath) {
4         this.filePath = filePath;
5     }
```


```

6     abstract public void accept(Extractor extractor);
7 }
8
9 public class PdfFile extends ResourceFile {
10     public PdfFile(String filePath) {
11         super(filePath);
12     }
13
14     @Override
15     public void accept(Extractor extractor) {
16         extractor.extract2txt(this);
17     }
18
19     //...
20 }
21
22 //...PPTFile、WordFile跟PdfFile类似，这里就省略了...
23 //...Extractor代码不变...
24
25 public class ToolApplication {
26     public static void main(String[] args) {
27         Extractor extractor = new Extractor();
28         List<ResourceFile> resourceFiles = listAllResourceFiles(args[0]);
29         for (ResourceFile resourceFile : resourceFiles) {
30             resourceFile.accept(extractor);
31         }
32     }
33
34     private static List<ResourceFile> listAllResourceFiles(String resourceDirectory)
35     {
36         List<ResourceFile> resourceFiles = new ArrayList<>();
37         //...根据后缀(pdf/ppt/word)由工厂方法创建不同的类对象(PdfFile/PPTFile/WordFile)
38         resourceFiles.add(new PdfFile("a.pdf"));
39         resourceFiles.add(new WordFile("b.word"));
40         resourceFiles.add(new PPTFile("c.ppt"));
41         return resourceFiles;
42     }
43 }

```

在执行第 30 行的时候，根据多态特性，程序会调用实际类型的 accept 函数，比如 PdfFile 的 accept 函数，也就是第 16 行代码。而 16 行代码中的 this 类型是 PdfFile 的，在编译的时候就确定了，所以会调用 extractor 的 extract2txt(PdfFile pdfFile) 这个重载函数。这个实现思路是不是很有技巧？这是理解访问者模式的关键所在，也是我之前所说的访问者模式不好理解的原因。

现在，如果要继续添加新的功能，比如前面提到的压缩功能，根据不同的文件类型，使用不同的压缩算法来压缩资源文件，那我们该如何实现呢？我们需要实现一个类似 Extractor 类的新类 Compressor 类，在其中定义三个重载函数，实现对不同类型资源文件的压缩。除此之外，我们还要在每个资源文件类中定义新的 accept 重载函数。具体的代码如下所示：

 复制代码

```
1 public abstract class ResourceFile {
2     protected String filePath;
3     public ResourceFile(String filePath) {
4         this.filePath = filePath;
5     }
6     abstract public void accept(Extractor extractor);
7     abstract public void accept(Compressor compressor);
8 }
9
10 public class PdfFile extends ResourceFile {
11     public PdfFile(String filePath) {
12         super(filePath);
13     }
14
15     @Override
16     public void accept(Extractor extractor) {
17         extractor.extract2txt(this);
18     }
19
20     @Override
21     public void accept(Compressor compressor) {
22         compressor.compress(this);
23     }
24
25     //...
26 }
27 }
28 //...PPTFile、WordFile跟PdfFile类似，这里就省略了...
29 //...Extractor代码不变
30
31 public class ToolApplication {
32     public static void main(String[] args) {
33         Extractor extractor = new Extractor();
34         List<ResourceFile> resourceFiles = listAllResourceFiles(args[0]);
35         for (ResourceFile resourceFile : resourceFiles) {
36             resourceFile.accept(extractor);
37         }
38
39         Compressor compressor = new Compressor();
40         for(ResourceFile resourceFile : resourceFiles) {
```


```

41     resourceFile.accept(compressor);
42 }
43 }
44
45 private static List<ResourceFile> listAllResourceFiles(String resourceDirectory
46     List<ResourceFile> resourceFiles = new ArrayList<>();
47     //...根据后缀(pdf/ppt/word)由工厂方法创建不同的类对象(PdfFile/PPTFile/WordFile)
48     resourceFiles.add(new PdfFile("a.pdf"));
49     resourceFiles.add(new WordFile("b.word"));
50     resourceFiles.add(new PPTFile("c.ppt"));
51     return resourceFiles;
52 }
53 }

```

上面代码还存在一些问题，添加一个新的业务，还是需要修改每个资源文件类，违反了开闭原则。针对这个问题，我们抽象出来一个 Visitor 接口，包含三个命名非常通用的 visit() 重载函数，分别处理三种不同类型的资源文件。具体做什么业务处理，由实现这个 Visitor 接口的具体的类来决定，比如 Extractor 负责抽取文本内容，Compressor 负责压缩。当我们新添加一个业务功能的时候，资源文件类不需要做任何修改，只需要修改 ToolApplication 的代码就可以了。

按照这个思路我们可以对代码进行重构，重构之后的代码如下所示：

 复制代码

```

1 public abstract class ResourceFile {
2     protected String filePath;
3     public ResourceFile(String filePath) {
4         this.filePath = filePath;
5     }
6     abstract public void accept(Visitor visitor);
7 }
8
9 public class PdfFile extends ResourceFile {
10     public PdfFile(String filePath) {
11         super(filePath);
12     }
13
14     @Override
15     public void accept(Visitor visitor) {
16         visitor.visit(this);
17     }

```



```
18     //...
19 }
20 //...PPTFile、WordFile跟PdfFile类似, 这里就省略了...
21
22 public interface Visitor {
23     void visit(PdfFile pdfFile);
24     void visit(PPTFile pdfFile);
25     void visit(WordFile pdfFile);
26 }
27
28 public class Extractor implements Visitor {
29     @Override
30     public void visit(PPTFile pptFile) {
31         //...
32         System.out.println("Extract PPT.");
33     }
34
35     @Override
36     public void visit(PdfFile pdfFile) {
37         //...
38         System.out.println("Extract PDF.");
39     }
40
41     @Override
42     public void visit(WordFile wordFile) {
43         //...
44         System.out.println("Extract WORD.");
45     }
46 }
47
48 public class Compressor implements Visitor {
49     @Override
50     public void visit(PPTFile pptFile) {
51         //...
52         System.out.println("Compress PPT.");
53     }
54
55     @Override
56     public void visit(PdfFile pdfFile) {
57         //...
58         System.out.println("Compress PDF.");
59     }
60
61     @Override
62     public void visit(WordFile wordFile) {
63         //...
64         System.out.println("Compress WORD.");
65     }
66 }
```

```

67 }
68
69 public class ToolApplication {
70     public static void main(String[] args) {
71         Extractor extractor = new Extractor();
72         List<ResourceFile> resourceFiles = listAllResourceFiles(args[0]);
73         for (ResourceFile resourceFile : resourceFiles) {
74             resourceFile.accept(extractor);
75         }
76
77         Compressor compressor = new Compressor();
78         for (ResourceFile resourceFile : resourceFiles) {
79             resourceFile.accept(compressor);
80         }
81     }
82
83     private static List<ResourceFile> listAllResourceFiles(String resourceDirectory) {
84         List<ResourceFile> resourceFiles = new ArrayList<>();
85         //... 根据后缀(pdf/ppt/word)由工厂方法创建不同的类对象(PdfFile/PPTFile/WordFile)
86         resourceFiles.add(new PdfFile("a.pdf"));
87         resourceFiles.add(new WordFile("b.word"));
88         resourceFiles.add(new PPTFile("c.ppt"));
89         return resourceFiles;
90     }
91 }
92

```

## 重新来看访问者模式

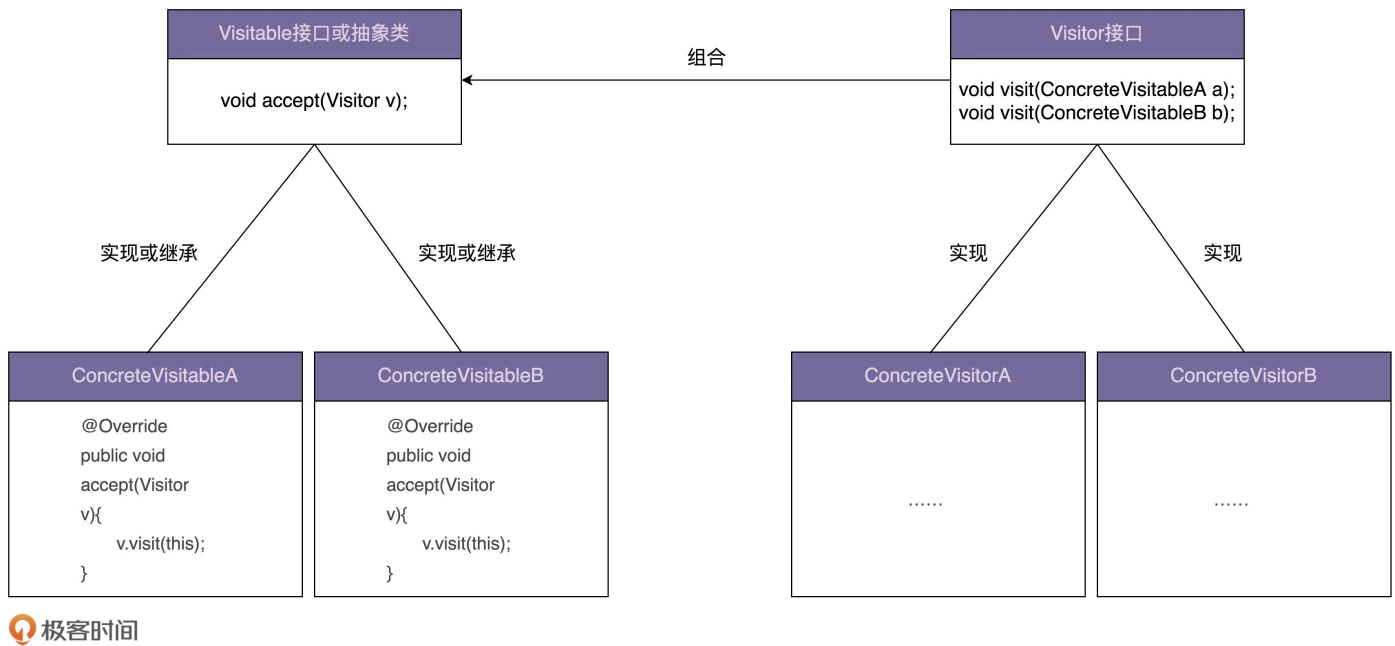
刚刚我带你一步一步还原了访问者模式诞生的思维过程，现在，我们回过头来总结一下，这个模式的原理和代码实现。

访问者者模式的英文翻译是 Visitor Design Pattern。在 GoF 的《设计模式》一书中，它是这么定义的：

Allows for one or more operation to be applied to a set of objects at runtime, decoupling the operations from the object structure.

翻译成中文就是：允许一个或者多个操作应用到一组对象上，解耦操作和对象本身。

定义比较简单，结合前面的例子不难理解，我就不过多解释了。对于访问者模式的代码实现，实际上，在上面例子中，经过层层重构之后的最终代码，就是标准的访问者模式的实现代码。这里，我又总结了一张类图，贴在了下面，你可以对照着前面的例子代码一块儿来看一下。



最后，我们再来看下，访问者模式的应用场景。

一般来说，访问者模式针对的是一组类型不同的对象（PdfFile、PPTFile、WordFile）。不过，尽管这组对象的类型是不同的，但是，它们继承相同的父类（ResourceFile）或者实现相同的接口。在不同的应用场景下，我们需要对这组对象进行一系列不相关的业务操作（抽取文本、压缩等），但为了避免不断添加功能导致类（PdfFile、PPTFile、WordFile）不断膨胀，职责越来越不单一，以及避免频繁地添加功能导致的频繁代码修改，我们使用访问者模式，将对象与操作解耦，将这些业务操作抽离出来，定义在独立细分的访问者类（Extractor、Compressor）中。

## 重点回顾

好了，今天的内容到此就讲完了。我们一块来总结回顾一下，你需要重点掌握的内容。

访问者模式允许一个或者多个操作应用到一组对象上，设计意图是解耦操作和对象本身，保持类职责单一、满足开闭原则以及应对代码的复杂性。

对于访问者模式，学习的主要难点在代码实现。而代码实现比较复杂的主要原因是，函数重载在大部分面向对象编程语言中是静态绑定的。也就是说，调用类的哪个重载函数，是在编译期间，由参数的声明类型决定的，而非运行时，根据参数的实际类型决定的。

正是因为代码实现难理解，所以，在项目中应用这种模式，会导致代码的可读性比较差。如果你的同事不了解这种设计模式，可能就会读不懂、维护不了你写的代码。所以，除非不得已，不要使用这种模式。

## 课堂讨论

实际上，今天举的例子不用访问者模式也可以搞定，你能够想到其他实现思路吗？

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。

### AI智能总结

本文深入讲解了访问者模式的设计思想和实现方式，旨在帮助读者快速了解访问者模式的概念和应用。通过手把手地带领读者还原访问者模式诞生的思维过程，让读者切身感受到创造一种新的设计模式并不是难事。文章通过代码示例展示了访问者模式的演进思路，重点讲解了如何利用访问者模式解决代码扩展性和灵活性的问题。总的来说，本文通过深入的讲解和实际代码演示，帮助读者理解访问者模式的设计思想和实现方式，为读者提供了一种快速理解和应用访问者模式的方法。同时，作者也提到了访问者模式的应用场景和难点，以及建议在项目中谨慎使用这种模式。

© 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 (95)

最新 精选



DFighting

2020-11-22

看完这个访问者模式，我第一反应是和桥接模式好相似，都是好像是操作与数据独立扩展，但是桥接模式主要使用的是组合，他的数据结构不支持扩展，进而我就理解了访问者模式和桥接模式不同的地方：桥接是对固定的数据结合进行多维度的独立扩展，每个维度的扩展可以在使用的时候随意组合，但是数据结构不支持改变，因为单纯的组合模式实现不了这种静态的“多态”；但是访问者模式最关键的地方是vist(this)，这么实现就做到了数据和操作两个的独立扩展，有新增的数据类型或者操作的时候都只需要按需扩展数据结构和操作即可，虽然这会涉及

到所有的操作类，但是这并未对已存在的功能做出影响，是符合开闭原则的，但是这个扩展操作因为需要支持多种数据结构，所以不适合如桥接模式这种多维度独立扩展，因为那样需要改动很多的类和代码，不合适

作者回复: 嗯嗯

共 4 条评论 >

👍 9



起个名字好难

2020-11-29

把属性和行为分离的前提是抽象，如果要访问的对象结构一旦改变必然是灾难。在我看来最困难的地方还是抽象。我在阅读jsqlparser源码的时候发现代码相对容易理解，但是在抽象sql时，真的感觉类都要爆炸了，没有一定的抽象能力真的搞不定

作者回复: 嗯嗯 加油



CoderArthur

2020-11-22

很nice的设计模式，差点略过没看，还好点进来再看第二遍，第二遍略懂了点。

昨天在实现迭代器模式的时候，也碰到过运行时期动态选择和编译器静态选择的问题，现在按照作者的写法思考下怎么解决我的问题。

作者回复: 是有点不好理解



test

2020-04-08

访问者模式解决的痛点主要是需要动态绑定的类型，所以调用哪个重载版本，其参数中的子类必须传入静态类型为目标子类的参数，并在方法中使用传入参数的动态绑定。如果不使用访问者模式，可以使用策略模式，使用工厂模式在map中保存type和具体子类实例的映射，在使用的时候，根据type的不同调用不同子类的方法（动态绑定）。

共 4 条评论 >

👍 65



Jxin

2020-04-08

- 1.虽然策略模式也能实现，但这个场景用访问者模式其实会优雅很多。
- 2.因为多种类型的同个操作聚合在了一起，那么因为这些类型是同父类的，所以属于父类的一些相同操作就能抽私有共用方法。
- 3.而策略模式，因为各个类型的代码都分割开了，那么就只好复制黏贴公共部分了。
- 4.另外，写合情合理的优雅代码，然后别人看不懂，一顿吹也是极爽的。只是一般节奏都挺快，第一时间可能就是策略模式走你，然后就没有然后了。

共 8 条评论 >

👍 41



李小四

2020-04-08

设计模式\_68:

# 作业:

今天的需求，我的第一反映是策略模式。

# 感想:

挺认同文章的观点，别人写了这种模式要看得懂，自己还是不要用比较好。

给转述师提个Tip: 程序开发中常常用数字`2`代替`to`、用数字`4`代替`for`,比如文中的`extract2txt`，这时要读作`extract to txt`，而不是`extract 2(中文读音er) txt`。

共 12 条评论 >

👍 19



Liam

2020-04-08

antlr(编译器框架)对语法树进行解析的时候就是通过visitor模式实现了扩展

共 1 条评论 >

👍 15



小晏子

2020-04-08

课后思考：可以使用策略模式，对于不同的处理方式定义不同的接口，然后接口中提供对于不同类型文件的实现，再使用静态工厂类保存不同文件类型和不同处理方法的映射关系。对于后续扩展的新增文件处理方法，比如composer，按同样的方式实现一组策略，然后修改application代码使用对应的策略。

共 1 条评论 >

👍 12



写代码的

2020-09-13

当一组重载函数的参数类型是继承同一个接口或者父类的话，如果传入的参数的静态类型是这个接口或者父类，java是无法决定使用哪个重载函数的。访问者模式的巧妙之处在于，我们可以借助多态，利用多态的动态分派特性，让这个参数暴露一个方法，使得这一组重载函数（或者说声明这组重载函数的类）能进入参数对象内部（也就是让参数暴露一个参数类型是这组重载函数所在类的方法）。一旦这组重载函数进入了参数对象内部，这组重载函数就知道了它的真实类型了，这个时候再调用重载函数，就能根据参数的具体类型找到合适的重载方法了。用一个通俗的例子来理解。敌军有多种类型的基地，我军有一套破坏敌军各种类型基地的方案，但是敌军的基地经过伪装，看上去都一样，我军在外面是无法决定使用哪套方案来攻击基地的。幸运的是，我军了解到，敌军每天中午会允许一批物资车进入基地，于是我方军队伪装成了敌军物资车进入到了敌军内部。进入基地之后我军了解了其真实类型，于是我军根据基地真实类型选择了相应的攻击方案，将敌军基地摧毁。

共 1 条评论 >

👍 10



Frank

2020-04-11

如果不使用访问者模式，也许这也是一种改造方法：在Extractor类种在定义一个重载的方法，形参的类型为：ResourceFile，在该方法种判断参数的实际类型后再做分派。如下所示

共 5 条评论 >

👍 8