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

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，你会怎么来做呢？

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。

```
public abstract class ResourceFile {
  protected String filePath;

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

  public abstract void extract2txt();
}

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

  @Override
  public void extract2txt() {
    //...省略一大坨从PPT中抽取文本的代码...
    //...将抽取出来的文本保存在跟filePath同名的.txt文件中...
    System.out.println("Extract PPT.");
  }
}

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

  @Override
```

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

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

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

// 运行结果是:
// Extract PDF.
// Extract WORD.
// Extract PPT.
public class ToolApplication {
  public static void main(String[] args) {
    List<ResourceFile> resourceFiles = listAllResourceFiles(args[0]);
    for (ResourceFile resourceFile : resourceFiles) {
      resourceFile.extract2txt();
    }
  }

  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;
  }
}
```

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。

```
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;
  }
}
```

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。

```
public abstract class ResourceFile {
  protected String filePath;
  public ResourceFile(String filePath) {
    this.filePath = filePath;
  }
  abstract public void accept(Extractor extractor);
}

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

  @Override
  public void accept(Extractor extractor) {
    extractor.extract2txt(this);
  }

  //...
}

//...PPTFile、WordFile跟PdfFile类似，这里就省略了...
//...Extractor代码不变...

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

  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;
  }
}
```

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：

```
public abstract class ResourceFile {
  protected String filePath;
  public ResourceFile(String filePath) {
    this.filePath = filePath;
  }
  abstract public void accept(Extractor extractor);
  abstract public void accept(Compressor compressor);
}

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

  @Override
  public void accept(Extractor extractor) {
    extractor.extract2txt(this);
  }

  @Override
  public void accept(Compressor compressor) {
    compressor.compress(this);
  }

  //...
}
}
//...PPTFile、WordFile跟PdfFile类似，这里就省略了...
//...Extractor代码不变

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

    Compressor compressor = new Compressor();
    for(ResourceFile resourceFile : resourceFiles) {
      resourceFile.accept(compressor);
    }
  }

  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;
  }
}
```

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，包含是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：

```
public abstract class ResourceFile {
  protected String filePath;
  public ResourceFile(String filePath) {
    this.filePath = filePath;
  }
  abstract public void accept(Visitor vistor);
}

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

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

  //...
}
//...PPTFile、WordFile跟PdfFile类似，这里就省略了...

public interface Visitor {
  void visit(PdfFile pdfFile);
  void visit(PPTFile pdfFile);
  void visit(WordFile pdfFile);
}

public class Extractor implements Visitor {
  @Override
  public void visit(PPTFile pptFile) {
    //...
    System.out.println("Extract PPT.");
  }

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

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

public class Compressor implements Visitor {
  @Override
  public void visit(PPTFile pptFile) {
    //...
    System.out.println("Compress PPT.");
  }

  @Override
```

```
  public void visit(PdfFile pdfFile) {
    //...
    System.out.println("Compress PDF.");
  }

  @Override
  public void visit(WordFile wordFile) {
    //...
    System.out.println("Compress WORD.");
  }

}

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

    Compressor compressor = new Compressor();
    for(ResourceFile resourceFile : resourceFiles) {
      resourceFile.accept(compressor);
    }
  }

  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;
  }
}
```

重新来看访问者模式

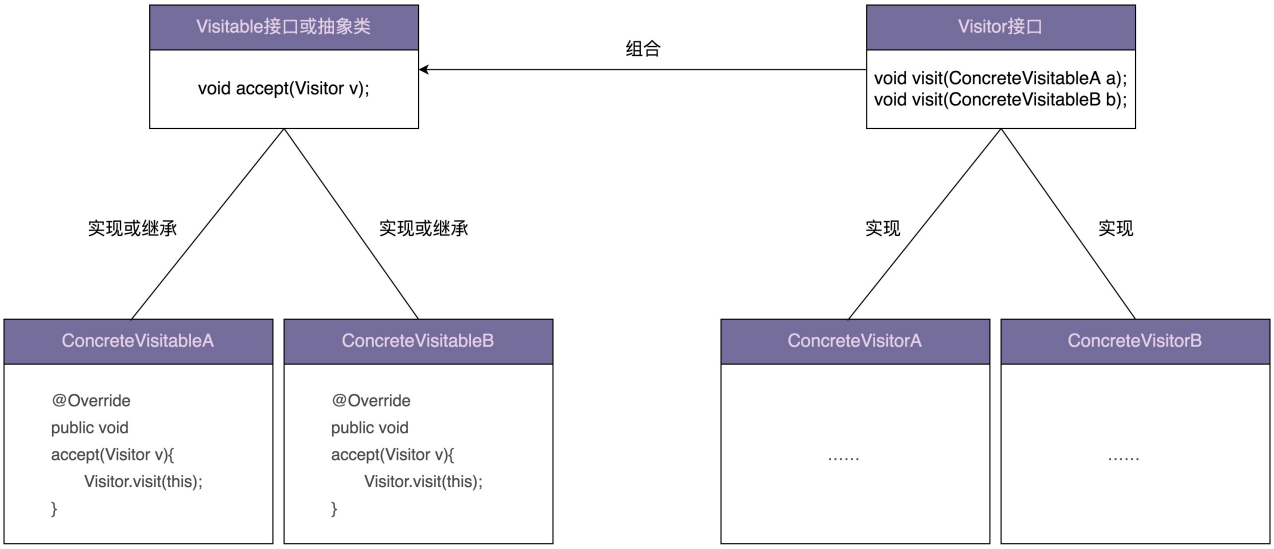
刚刚我带你一步一步还原了访问者模式诞生的思维过程，现在，我们回过头来总结一下，这个模式的原理和代码实现。

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。

精选留言：

- Monday 2020-04-08 08:41:48
  独立细分的访问者类（Extractor、Compressor），这些类分别对应所有类型对象某一操作的实现，如果类型多了，这些访问者类也会爆炸。 [5赞]

- test 2020-04-08 12:30:17
  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）。 [3赞]

- 李小四 2020-04-08 10:41:58
  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`。 [2赞]

- 小晏子 2020-04-08 09:05:37
  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。 [2赞]

- Liam 2020-04-08 08:57:08
  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 [1赞]

- Frank 2020-04-08 08:10:40
  打卡 今日学习访问者设计模式，收获如下：
  访问者模式表示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。体现了SRP原则。这个原则的代码实现比较复杂，关键要理解“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”。
  文章的例子通过4个版本的迭代，从一个最小原型实现，逐渐重构成一个符合访问者模式的代码。重构过程中不断使用前面讲解到的单一职责原则，基于接口而非实现编程等原则，使用工厂模式等来使代码逐渐符合“高内聚，低耦合”的目标。在实际开发中，该模式理解起来难度大，不建议使用。 [1赞]

- 守拙 2020-04-09 21:45:17
  课堂讨论:
  使用策略+静态工厂可以实现需求.
  访问者模式为一组对象执行同一操作, 很容易想到每种对象对应一种操作策略.

  ```
  ResourceFile pdf = new PdfFile();
  ResourceFile ppt = new PptFile();
  ResourceFile word = new WordFile();

  String s = ExtractStrategyFactory.create(pdf.getClass()).extractToTxt(pdf);

  //工厂实现
  public class ExtractStrategyFactory {

  private ExtractStrategyFactory(){}
  ```

```
private static Map<Class<? extends ResourceFile>, ExtractStrategy> map = new HashMap<>();
static {
map.put(PdfFile.class, new PdfExtractStrategy());
map.put(PptFile.class, new PptExtractStrategy());
map.put(WordFile.class, new WordExtractStrategy());
}

public static ExtractStrategy create(Class<? extends ResourceFile> clazz){
return map.get(clazz);
}
}
```

- 筱乐乐哦 2020-04-09 01:39:51
  想问下这种设计模式在哪个的源码中有使用咋？先看下是怎么用的，老师写的例子还是比较简单易懂的，但感觉实际用的时候，需要根据业务抽象，就不好弄了呀

- L🚲🐱 2020-04-08 22:02:15
  可以使用策略模式, 用工厂方法来

- QQ怪 2020-04-08 16:33:37
  问下老师，有没有哪个优秀的框架用到这个模式的?

- Jxin 2020-04-08 13:18:11
  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。

- Ken张云忠 2020-04-08 11:37:40
  文中提到重载是一种静态绑定,多态是一种动态绑定,这里多态可以理解为重写吧
  jvm中不存在重载,因为编译阶段时已经确定了目标方法.
  动态绑定需要根据方法描述符来确定目标方法,最基础的是方法表的方式,另外还有优化手段内联缓存和方法内联.

- Ken张云忠 2020-04-08 11:21:01
  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?
  可以使用策略模式.
  定义读取策略接口ExtractorStrategy并实现三个策略,再定义一个策略工厂类,以文件类型作为key,以对应策略实现作为value,使用时通过具体的ResourceFile类型获取对应的策略实现类型,然后再调用实现函数.

- Heaven 2020-04-08 11:02:23
  1.能,只需要应用策略模式,在策略类中提供一个入口,在入口方法中进行类的判断,调用即可,可读性较好,不过代码比较冗余
  2.对于本章,其实可看做就是不同类型的用户去访问一个服务器,根据不同的用户去做不同的处理.比较困难的就是用户可能不知道自己是谁,那就需要我们去帮他知道是谁

- ， 2020-04-08 10:48:40
  对方法绑定感兴趣的同学可以看 深入理解java虚拟机 书中的->虚拟机字节码执行引擎->方法调用->方法分派的章节,详细介绍了虚拟机是如何通过静态绑定和动态绑定,实现方法重载和方法重写的
  课后题:
  如果子类对象的行为是一样的,都具备同样的行为(抽取,压缩,分析),那么用模板方法模式也完全可以,修改的成本并没有多大
  如果子类对象的行为是不一致的,比如对ppt只进行抽取,对word进行抽取和压缩,那么各个子类都维护自己的方法比较好,没有必要抽取出来

- L 2020-04-08 09:08:30
  课后题，策略模式+抽象工厂模式