私塾在线《研磨设计模式》
——跟着CC学设计系列精品教程

本节课程概览

- 学习解释器模式

一：初识解释器模式

包括：定义、结构、参考实现

二：体会解释器模式

包括：场景问题、不用模式的解决方案、使用模式的解决方案

三：理解解释器模式

包括：认识解释器模式、读取多个元素或属性的值 、解析器 、解释器模式的优缺点

四：思考解释器模式

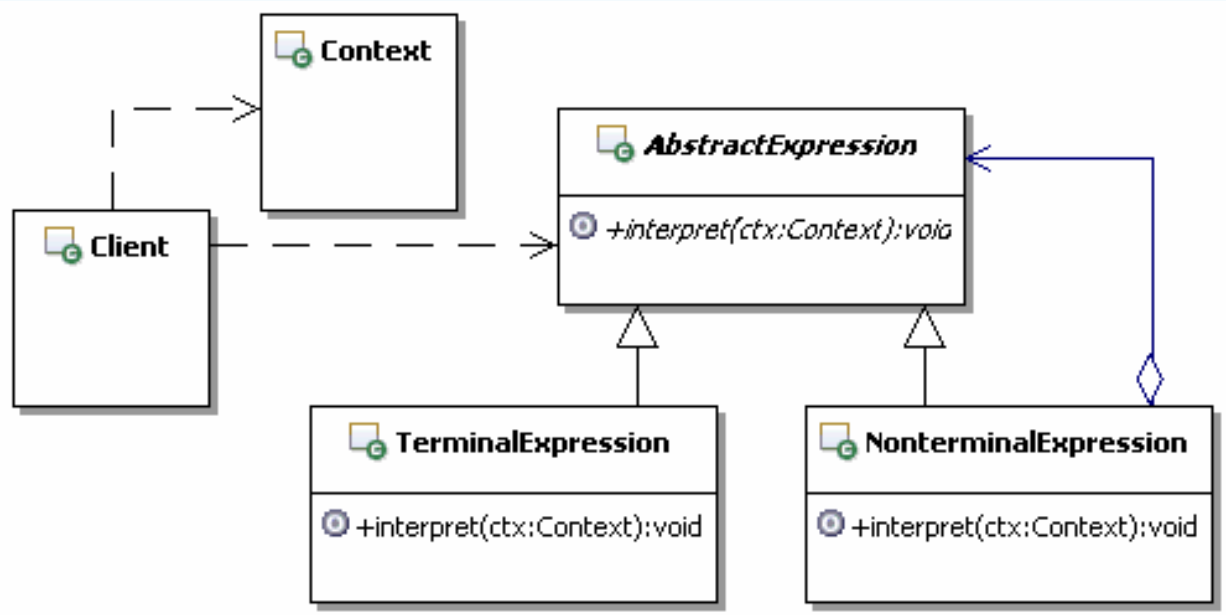
包括：解释器模式的本质、何时选用

初识解释器模式

- 定义

  给定一个语言，定义它的文法的一种表示，并定义一个解释器，这个解释器使用该表示来解释语言中的句子。

- 结构和说明



体会解释器模式

AbstractExpression：定义解释器的接口，约定解释器的解释操作。

TerminalExpression：

终结符解释器，用来实现语法规则中和终结符相关的操作，不再包含其它的解释器，如果用组合模式来构建抽象语法树的话，就相当于组合模式中的叶子对象，可以有多种终结符解释器。

NonterminalExpression：

非终结符解释器，用来实现语法规则中非终结符相关的操作，通常一个解释器对应一个语法规则，可以包含其它的解释器，如果用组合模式来构建抽象语法树的话，就相当于组合模式中的组合对象，可以有多种非终结符解释器。

Context：

上下文，通常包含各个解释器需要的数据，或是公共的功能。

Client：

客户端，指的是使用解释器的客户端，通常在这里去把按照语言的语法做的表达式，转换成为使用解释器对象描述的抽象语法树，然后调用解释操作。

体会解释器模式

- 读取配置文件

考虑这样一个实际的应用，维护系统自定义的配置文件。几乎每个实际的应用系统都有与应用自身相关的配置文件，这个配置文件是由开发人员根据需要自定义的，系统运行时会根据配置的数据进行相应的功能处理。

系统现有的配置数据很简单，主要是JDBC所需要的数据，还有默认读取Spring的配置文件，目前系统只需要一个Spring的配置文件。示例如下：

```
<?xml version="1.0" encoding="UTF-8"?>
<root>
    <jdbc>
        <driver-class>驱动类名</driver-class>
        <url>连接数据库的URL</url>
        <user>连接数据库的用户名</user>
        <password>连接数据库的密码</password>
    </jdbc>
    <application-xml>缺省读取的Spring配置的文件名称</application-xml>
</root>
```

现在的功能需求是：如何能够灵活的读取配置文件的内容？

体会解释器模式

- 不用模式的解决方案

不就是读取配置文件吗？实现很简单，直接读取并解析xml就可以了。读取xml的应用包很多，这里都不用，直接采用最基础的Dom解析就可以了。另外，读取到xml中的值过后，后续如何处理，这里也不去管，这里只是实现把配置文件读取并解析出来。

体会解释器模式

- 有何问题

看了上面的实现，多简单啊，就是最基本的Dom解析嘛，要是采用其它的开源工具包，比如dom4j、jDom之类的来处理，会更简单，这好像不值得一提呀，真的是这样吗？

请思考一个问题：如果配置文件的结构需要变动呢？仔细想想，就会感觉出问题来了。还是先看例子，然后再来总结这个问题。

随着开发的深入进行，越来越多可配置的数据被抽取出来，需要添加到配置文件中，比如与数据库的连接配置：就加入了是否需要、是否使用DataSource等配置。除了这些还加入了一些其它需要配置的数据，例如：系统管理员、日志记录方式、缓存线程的间隔时长、默认读取哪些Spring配置文件等等，示例如下：

体会解释器模式

```
<?xml version="1.0" encoding="UTF-8"?>
<root>
    <database-connection>
        <connection-type>连接数据库的类型,1-用Spring集成的方式
            (也就是不用下面两种方式了),2-DataSource（就是使用JNDI）,
            3-使用JDBC自己来连接数据库
        </connection-type>
        <jndi>DataSource的方式用，服务器数据源的JNDI名称</jndi>
        <jdbc>跟上面一样，省略了</jdbc>
    </database-connection>
    <system-operator>系统管理员ID</system-operator>
    <log>
        <operate-type>记录日志的方式,1-数据库，2-文件</operate-type>
        <file-name>记录日志的文件名称</file-name>
    </log>
    <thread-interval>缓存线程的间隔时长</thread-interval>
    <spring-default>
        <application-xmls>
            <application-xml>
                缺省读取的Spring配置的文件名称
            </application-xml>
            <application-xml>
                其它需要读取的Spring配置的文件名称
            </application-xml>
        </application-xmls>
    </spring-default>
</root>
```

体会解释器模式

改变一下配置文件不是件大事情，但是带来的一系列麻烦也不容忽视，比如：修改了配置文件的结构，那么读取配置文件的程序就需要做出相应的变更；用来封装配置文件数据的数据对象也需要相应的修改；外部使用配置文件的地方，获取数据的地方也会相应变动。

当然在这一系列麻烦中，最让人痛苦的莫过于修改读取配置文件的程序了，有时候几乎是重写。比如在使用Dom读取第一个配置文件，读取默认的Spring配置文件的值的时候，可能的片断代码示例如下：

```
//获取application-xml
NodeList applicationXmlNode = 
                doc.getElementsByTagName("application-xml");
String applicationXml = applicationXmlNode.item(0)
                .getFirstChild().getNodeValue();
System.out.println("applicationXml=="+applicationXml);
```

体会解释器模式

但是如果配置文件改成第二个，文件的结构发生了改变，需要读取的配置文件变成了多个了，读取的程序也发生了改变，而且application-xml节点也不是直接从doc下获取了。几乎是完全重写了，此时可能的片断代码示例如下：

```
//先要获取spring-default，然后获取application-xmls
//然后才能获取application-xml
NodeList springDefaultNode =
                doc.getElementsByTagName("spring-default");
NodeList appXmlsNode = ((Element)springDefaultNode.item(0))
                .getElementsByTagName("application-xmls");
NodeList appXmlNode = ((Element)appXmlsNode.item(0))
                .getElementsByTagName("application-xml");
//循环获取每个application-xml元素的值
for(int i=0;i<appXmlNode.getLength();i++){
    String applicationXml = appXmlNode.item(i)
                .getFirstChild().getNodeValue();
    System.out.println("applicationXml=="+applicationXml);
}
```

体会解释器模式

仔细对比上面在xml变化前后读取值的代码，你会发现，由于xml结构的变化，导致读取xml文件内容的代码，基本上完全重写了。

问题还不仅仅限于读取元素的值，同样体现在读取属性上。可能有些朋友说可以换不同的xml解析方式来简化，不是还有Sax解析，实在不行换用其它开源的解决方案。

确实通过使用不同的解析xml的方式是会让程序变得简单点，但是每次xml的结构发生变化过后，或多或少都是需要修改程序中解析xml部分的。

有没有办法解决这个问题呢？也就是当xml的结构发生改变过后，能够很方便的获取相应元素、或者是属性的值，而不用再去修改解析xml的程序。

体会解释器模式

- 使用模式来解决的思路

要想解决当xml的结构发生改变后，不用修改解析部分的代码，一个自然的思路就是要把解析部分的代码写成公共的，而且还要是通用的，能够满足各种xml取值的需要，比如：获取单个元素的值，获取多个相同名称的元素的值，获取单个元素的属性的值，获取多个相同名称的元素的属性的值，等等。

要写成通用的代码，又有几个问题要解决，如何组织这些通用的代码？如何调用这些通用的代码？以何种方式来告诉这些通用代码，客户端的需要？

要解决这些问题，其中的一个解决方案就是解释器模式。在描述这个模式的解决思路之前，先解释两个概念，一个是解析器（不是指xml的解析器），一个是解释器。

1：这里的解析器，指的是把描述客户端调用要求的表达式，经过解析，形成一个抽象语法树的程序，不是指xml的解析器。

2：这里的解释器，指的是解释抽象语法树，并执行每个节点对应的功能的程序。

体会解释器模式

要解决通用解析xml的问题，第一步：需要先设计一个简单的表达式语言，在客户端调用解析程序的时候，传入用这个表达式语言描述的一个表达式，然后把这个表达式通过解析器的解析，形成一个抽象的语法树。

第二步：解析完成后，自动调用解释器来解释抽象语法树，并执行每个节点所对应的功能，从而完成通用的xml解析。

这样一来，每次当xml结构发生了更改，也就是在客户端调用的时候，传入不同的表达式即可，整个解析xml过程的代码都不需要再修改了。

体会解释器模式

- 为表达式设计简单的文法

为了通用，用root表示根元素，a、b、c、d等来代表元素，一个简单的xml如下：

```
<?xml version="1.0" encoding="UTF-8"?>
<root id="rootId">
    <a>
        <b>
            <c name="testC">12345</c>
            <d id="1">d1</d>
            <d id="2">d2</d>
            <d id="3">d3</d>
            <d id="4">d4</d>
        </b>
    </a>
</root>
```

体会解释器模式

- 约定表达式的文法如下：

1：获取单个元素的值：从根元素开始，一直到想要获取值的元素，元素中间用“/”分隔，根元素前不加“/”。比如表达式“root/a/b/c”就表示获取根元素下、a元素下、b元素下的c元素的值

2：获取单个元素的属性的值：要获取值的属性一定是表达式的最后一个元素的属性，在最后一个元素后面添加“.”然后再加上属性的名称。如“root/a/b/c.name”就表示获取根元素下、a元素下、b元素下、c元素的name属性的值

3：获取相同元素名称的值，当然是多个：要获取值的元素一定是表达式的最后一个元素，在最后一个元素后面添加“$”。比如表达式“root/a/b/d$”就表示获取根元素下、a元素下、b元素下的多个d元素的值的集合

4：获取相同元素名称的属性的值，当然也是多个：要获取属性值的元素一定是表达式的最后一个元素，在最后一个元素后面添加“$”，然后在后面添加“.”然后再加上属性的名称，在属性名称后面也添加“$”。比如表达式“root/a/b/d$.id$”就表示获取根元素下、a元素下、b元素下的多个d元素的id属性的值的集合

体会解释器模式

- 示例说明

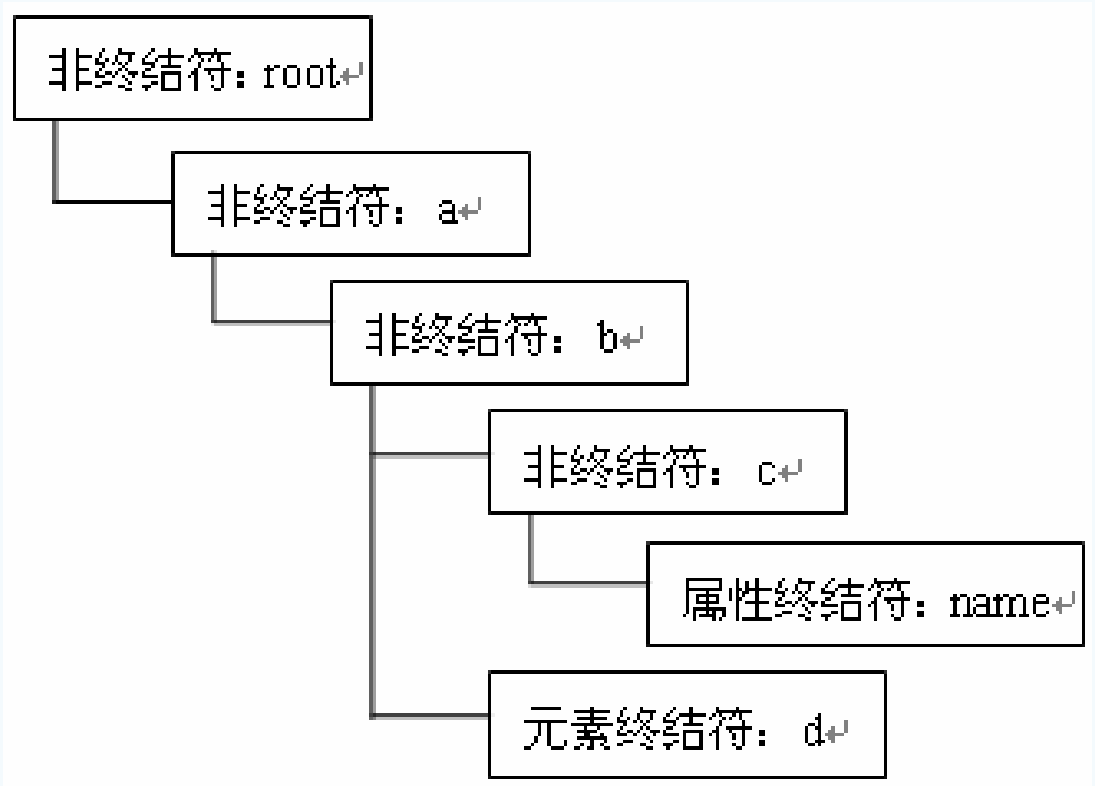
为了示例的通用性，就使用上面这个xml来实现功能，不去使用前面定义的具体的xml了，解决的方法是一样的。

另外一个问题，解释器模式主要解决的是“解释抽象语法树，并执行每个节点所对应的功能”，并不包含如何从一个表达式转换成为抽象的语法树。因此下面的范例就先来实现解释器模式所要求的功能。至于如何从一个表达式转换成为相应的抽象语法树，后面会给出一个示例。

对于抽象的语法树这个树状结构，很明显可以使用组合模式来构建。解释器模式把需要解释的对象分成了两大类，一类是节点元素，就是可以包含其它元素的组合元素，比如非终结符元素，对应成为组合模式的Composite；另一类是终结符元素，相当于组合模式的叶子对象。解释整个抽象语法树的过程，也就是执行相应对象的功能的过程。

体会解释器模式

比如上面的xml，对应成为抽象语法树，可能的结构如下图



理解解释器模式

- 认识解释器模式

1：解释器模式的功能

解释器模式使用解释器对象来表示和处理相应的语法规则，一般一个解释器处理一条语法规则。理论上来说，只要能用解释器对象把符合语法的表达式表示出来，而且能够构成抽象的语法树，那都可以使用解释器模式来处理。

2：语法规则和解释器

语法规则和解释器之间是有对应关系的，一般一个解释器处理一条语法规则，但是反过来并不成立，一条语法规则是可以有多种解释和处理的，也就是一条语法规则可以对应多个解释器对象。

理解解释器模式

3：上下文的公用性

上下文在解释器模式中起到非常重要的作用，由于上下文会被传递到所有的解释器中，因此可以在上下文中存储和访问解释器的状态，比如前面的解释器可以存储一些数据在上下文中，后面的解释器就可以获取这些值。

另外还可以通过上下文传递一些在解释器外部，但是解释器需要的数据，也可以是一些全局的，公共的数据。

上下文还有一个功能，可以提供所有解释器对象的公共功能，类似于对象组合，而不是使用继承来获取公共功能，在每个解释器对象里面都可以调用。

理解解释器模式

4：谁来构建抽象语法树

在前面的示例中，大家已经发现，自己在客户端手工来构建抽象语法树，是很麻烦的，但是在解释器模式中，并没有涉及这部分功能，只是负责对构建好的抽象语法树进行解释处理。前面的测试简单，所以手工构建抽象语法树也不是特别困难的事，要是复杂了呢？如果还是手工创建，那跟修改解析xml的代码也差不了多少。后面会给大家讲提供解析器来把表达式转换成为抽象语法树。

还有一个问题，就是一条语法规则是可以对应多个解释器对象的，也就是说同一个元素，是可以转换成多个解释器对象的，这也就意味着同样一个表达式，是可以构成不同的抽象语法树的，这也造成构建抽象语法树变得很困难，而且工作量很大。

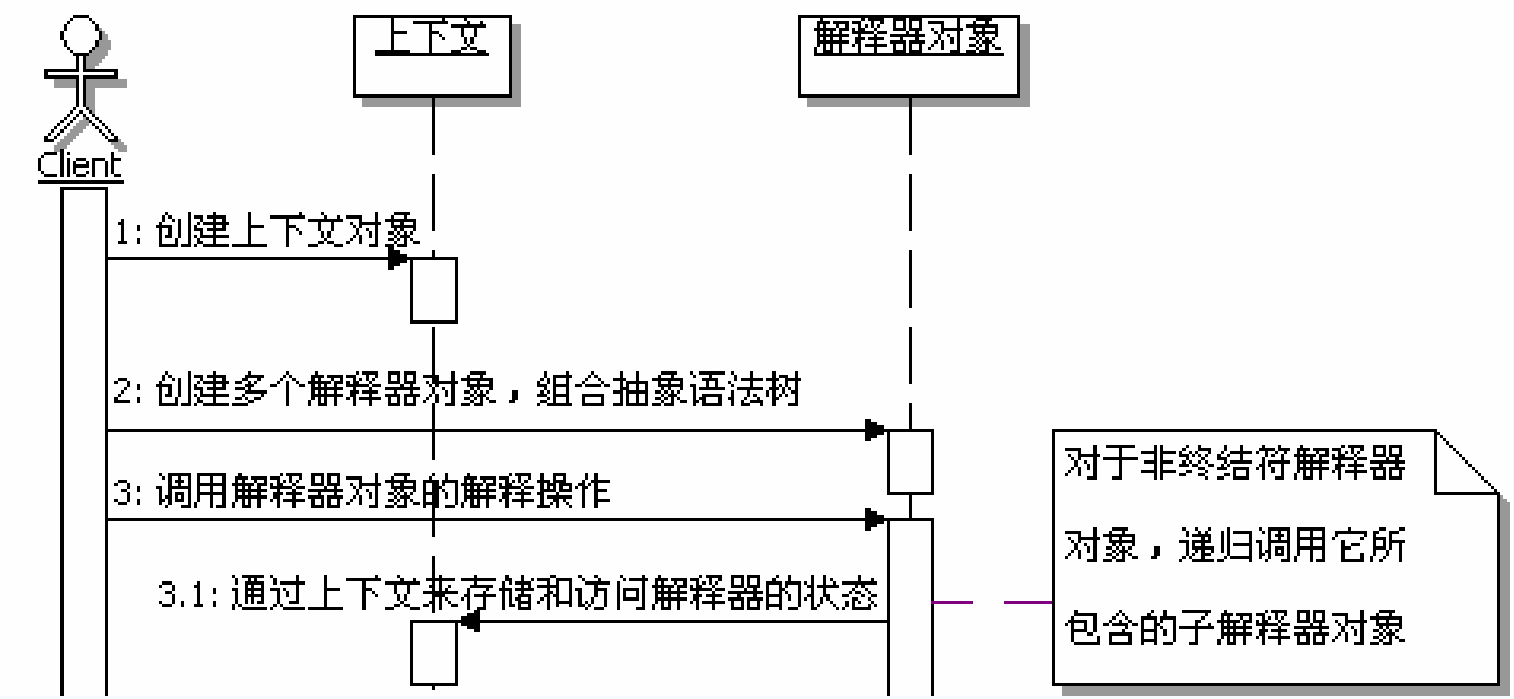
5：谁负责解释操作

只要定义好了抽象语法树，肯定是解释器来负责解释执行。虽然有不同的语法规则，但是解释器不负责选择究竟用哪一个解释器对象来解释执行语法规则，选择解释器的功能在构建抽象语法树的时候就完成了。

所以解释器只要忠实的按照抽象语法树解释执行就好了。

理解解释器模式

6：解释器模式的调用顺序示意图



理解解释器模式

- 读取多个元素或属性的值

获取多个值和前面获取单个值的实现思路大致相同，只是在取值的时候需要循环整个NodelList，依次取值，而不是只取出第一个来。当然，由于语法发生了变动，所以对应的解释器也需要发生改变。

首先是有了一个表示多个元素作为终结符的语法，比如“root/a/b/d$”中的“d$”；其次有了一个表示多个元素的属性作为终结符的语法，比如“root/a/b/d$.id$”中的“.id$”；最后还有一个表示多个元素，但不是终结符的语法，比如“root/a/b/d$.id$”中的“d$”。

理解解释器模式

- 解析器

解析器负责把表达式，解析转换成为解释器需要的抽象语法树。当然解析器是跟表达式的语法，还有解释器对象紧密关联的。

解析器有很多种实现方式，没有什么定式，只要能完成相应的功能即可，比如表驱动、语法分析生成程序等。这里的示例采用自己来分解表达式以实现构建抽象语法树的功能，没有使用递归，是用循环实现的。

1：实现思路

要实现解析器也不复杂，大约有下面三个步骤：

第一步：把客户端传递来的表达式进行分解，分解成为一个一个的元素，并用一个对应的解析模型来封装这个元素的一些信息。

第二步：根据每个元素的信息，转化成相对应的解析器对象

第三步：按照先后顺序，把这些解析器对象组合起来，得到抽象语法树

理解解释器模式

可能有朋友会说，为什么不把第一步和第二步合并，直接分解出一个元素就转换成相应的解析器对象呢？原因有两个：

其一是功能分离，不要让一个方法的功能过于复杂；

其二是为了今后的修改和扩展，现在语法简单，所以转换成解析器对象需要考虑的东西少，直接转换也不难，但要是语法复杂了，直接转换就很杂乱了。

事实上，封装解析属性的数据模型充当了第一步和第二步操作间的接口，使第一步和第二步都变简单了。

理解解释器模式

- 解析器示例小结

通过使用解释器模式，自行设计一种简单的语法，就可以用很简单的表达式来获取你想要的xml中的值了。有的朋友可能会想到XPath，没错，示例实现的功能就是类似于XPath的部分功能。

如果今后xml的结构要是发生了变化，或者是想要获取不同的值，基本上就是修改那个表达式而已，你可以试试看，能否完成前面实现过的功能。比如：

1：想要获取c元素的值，表达式为：“root/a/b/c”

2：想要获取c元素的name属性值，表达式为：“root/a/b/c.name”

3：想要获取d元素的值，表达式为：“root/a/b/d$”，获取d的属性上面已经测试了

理解解释器模式

- 解释器模式的优缺点
  1：易于实现语法
  2：易于扩展新的语法
  3：不适合复杂的语法

思考解释器模式

- 解释器模式的本质

  解释器模式的本质是：**分离实现，解释执行**

- 何时选用解释器模式

1：当有一个语言需要解释执行，并且可以将该语言中的句子表示为一个抽象语法树的时候，可以考虑使用解释器模式。

在使用解释器模式的时候，还有两个特点需要考虑，一个是语法相对应该比较简单，太复杂的语法不合适使用解释器模式；另一个是效率要求不是很高，对效率要求很高的情况下，不适合使用解释器模式。