链滴

Spring 框架中的设计模式（一）

作者：Jacklinsir

原文链接：https://ld246.com/article/1568097696354

来源网站：链滴

设计模式有助于遵循良好的编程实践。作为最流行的Web框架之一的Spring框架也使用其中的一些。

本文将介绍Spring Framework中使用的设计模式。这是5篇专题文章的第一部分。这次我们将发现Spring框架中使用的4种设计模式：解释器，构建器，工厂方法和抽象工厂。每部分将首先解释给定模式原理。紧接着，将会使用Spring的一个例子来加深理解。

解释器设计模式

在现实世界中，我们人类需要解释手势。他们可以对文化有不同的含义。这是我们的解释，给他们一意义。在编程中，我们还需要分析一件事情，并决定它是什么意思。我们可以用解释设计模式来做。

此模式基于表达式和评估器部分。第一个代表一个要分析的事情。这个分析是由评价者来做出的，它知道构成表达的人物的意义。不必要的操作是在一个上下文中进行的。

Spring主要以Spring Expression Language（SpEL）为例。这里快速提个醒，SpEL是一种由Spring org.springframework.expression.ExpressionParser实现分析和执行的语言。这些实现使用作为字串给出的Spel表达式，并将它们转换为org.springframework.expression.Expression的实例。上下组件由org.springframework.expression.EvaluationContext实现表示，例如：StandardEvaluation ontext。

举个SpEL的一个例子：

```
Writer writer = new Writer();

writer.setName("Writer's name");
StandardEvaluationContext modifierContext = new StandardEvaluationContext(subscriberCon
ext);
modifierContext.setVariable("name", "Overriden writer's name");
parser.parseExpression("name = #name").getValue(modifierContext);
System.out.println("writer's name is : " + writer.getName());
```

输出应打印“Overriden writer’ s name”。如你所见，一个对象的属性是通过一个表达式name = name进行修改的，这个表达式只有在ExpressionParser才能理解，因为提供了context（前面的样例的modifierContext实例）。

建设者模式

建设者设计模式是属于创建对象模式三剑客的第一种模式。该模式用于简化复杂对象的构造。要理解个概念，想象一个说明程序员简历的对象。在这个对象中，我们想存储个人信息（名字，地址等）以技术信息（知识语言，已实现的项目等）。该对象的构造可能如下所示：

```
// with constructor
Programmer programmer = new Programmer("first name", "last name", "address Street 39", "
IP code", "City", "Country", birthDateObject, new String[] {"Java", "PHP", "Perl", "SQL"}, new Str
ng[] {"CRM system", "CMS system for government"});
// or with setters
Programmer programmer = new Programmer();
programmer.setName("first name");
programmer.setLastName("last name");
// ... multiple lines after
programmer.setProjects(new String[] {"CRM system", "CMS system for government"});
```

Builder允许我们通过使用将值传递给父类的内部构建器对象来清楚地分解对象构造。所以对于我们这程序员简历的对象的创建，构建器可以看起来像：

```
public class BuilderTest {

  @Test
  public void test() {
    Programmer programmer = new Programmer.ProgrammerBuilder()
            .setFirstName("F").setLastName("L")
            .setCity("City").setZipCode("0000A").setAddress("Street 39")
            .setLanguages(new String[] {"bash", "Perl"})
            .setProjects(new String[] {"Linux kernel"}).build();
    assertTrue("Programmer should be 'F L' but was '" + programmer + "'",
        programmer.toString().equals("F L"));
  }

}

class Programmer {
  private String firstName;
  private String lastName;
  private String address;
  private String zipCode;
  private String city;
  private String[] languages;
  private String[] projects;

  private Programmer(String fName, String lName, String addr, String zip, String city, String[] l
ngs, String[] projects) {
    this.firstName = fName;
    this.lastName = lName;
    this.address = addr;
    this.zipCode = zip;
```

```
    this.city = city;
    this.languages = langs;
    this.projects = projects;
  }

  public static class ProgrammerBuilder {
    private String firstName;
    private String lastName;
    private String address;
    private String zipCode;
    private String city;
    private String[] languages;
    private String[] projects;

    public ProgrammerBuilder setFirstName(String firstName) {
      this.firstName = firstName;
      return this;
    }

    public ProgrammerBuilder setLastName(String lastName) {
      this.lastName = lastName;
      return this;
    }

    public ProgrammerBuilder setAddress(String address) {
      this.address = address;
      return this;
    }

    public ProgrammerBuilder setZipCode(String zipCode) {
      this.zipCode = zipCode;
      return this;
    }

    public ProgrammerBuilder setCity(String city) {
      this.city = city;
      return this;
    }

    public ProgrammerBuilder setLanguages(String[] languages) {
      this.languages = languages;
      return this;
    }
    public ProgrammerBuilder setProjects(String[] projects) {
      this.projects = projects;
      return this;
    }

    public Programmer build() {
      return new Programmer(firstName, lastName, address, zipCode, city, languages, projects);
    }
  }

  @Override
```

```
  public String toString() {
    return this.firstName + " "+this.lastName;
  }

}
```

可以看出，构建器后面隐藏了对象构造的复杂性，内部静态类接受链接方法的调用。在Spring中，我可以在org.springframework.beans.factory.support.BeanDefinitionBuilder类中检索这个逻辑。这一个允许我们以编程方式定义bean的类。我们可以在关于bean工厂后处理器的文章中看到它，BeanefinitionBuilder包含几个方法，它们为AbstractBeanDefinition抽象类的相关实现设置值，比如作域，工厂方法，属性等。想看看它是如何工作的，请查看以下这些方法：

```
public class BeanDefinitionBuilder {
      /**
    * The {@code BeanDefinition} instance we are creating.
    */
  private AbstractBeanDefinition beanDefinition;

  // ... some not important methods for this article

  // Some of building methods
  /**
    * Set the name of the parent definition of this bean definition.
    */
  public BeanDefinitionBuilder setParentName(String parentName) {
    this.beanDefinition.setParentName(parentName);
    return this;
  }

  /**
    * Set the name of the factory method to use for this definition.
    */
  public BeanDefinitionBuilder setFactoryMethod(String factoryMethod) {
    this.beanDefinition.setFactoryMethodName(factoryMethod);
    return this;
  }

  /**
    * Add an indexed constructor arg value. The current index is tracked internally
    * and all additions are at the present point.
    * @deprecated since Spring 2.5, in favor of {@link #addConstructorArgValue}
    */
  @Deprecated
  public BeanDefinitionBuilder addConstructorArg(Object value) {
    return addConstructorArgValue(value);
  }

  /**
    * Add an indexed constructor arg value. The current index is tracked internally
    * and all additions are at the present point.
    */
  public BeanDefinitionBuilder addConstructorArgValue(Object value) {
    this.beanDefinition.getConstructorArgumentValues().addIndexedArgumentValue(
                this.constructorArgIndex++, value);
```

```
    return this;
  }

  /**
   * Add a reference to a named bean as a constructor arg.
   * @see #addConstructorArgValue(Object)
   */
  public BeanDefinitionBuilder addConstructorArgReference(String beanName) {
    this.beanDefinition.getConstructorArgumentValues().addIndexedArgumentValue(
                this.constructorArgIndex++, new RuntimeBeanReference(beanName));
    return this;
  }

  /**
   * Add the supplied property value under the given name.
   */
  public BeanDefinitionBuilder addPropertyValue(String name, Object value) {
    this.beanDefinition.getPropertyValues().add(name, value);
    return this;
  }

  /**
   * Add a reference to the specified bean name under the property specified.
   * @param name the name of the property to add the reference to
   * @param beanName the name of the bean being referenced
   */
  public BeanDefinitionBuilder addPropertyReference(String name, String beanName) {
    this.beanDefinition.getPropertyValues().add(name, new RuntimeBeanReference(beanName)
;
    return this;
  }

  /**
   * Set the init method for this definition.
   */
  public BeanDefinitionBuilder setInitMethodName(String methodName) {
    this.beanDefinition.setInitMethodName(methodName);
    return this;
  }

  // Methods that can be used to construct BeanDefinition
  /**
   * Return the current BeanDefinition object in its raw (unvalidated) form.
   * @see #getBeanDefinition()
   */
  public AbstractBeanDefinition getRawBeanDefinition() {
    return this.beanDefinition;
  }

  /**
   * Validate and return the created BeanDefinition object.
   */
  public AbstractBeanDefinition getBeanDefinition() {
    this.beanDefinition.validate();
```

```
    return this.beanDefinition;
  }
}
```

工厂方法

创建对象模式三剑客的第二个成员是工厂方法设计模式。它完全适于使用动态环境作为Spring框架。际上，这种模式允许通过公共静态方法对象进行初始化，称为工厂方法。在这个概念中，我们需要定一个接口来创建对象。但是创建是由使用相关对象的类创建的。

但是在跳到Spring世界之前，让我们用Java代码做一个例子：

```
public class FactoryMethodTest {

  @Test
  public void test() {
    Meal fruit = Meal.valueOf("banana");
    Meal vegetable = Meal.valueOf("carrot");
    assertTrue("Banana should be a fruit but is "+fruit.getType(), fruit.getType().equals("fruit"));
    assertTrue("Carrot should be a vegetable but is "+vegetable.getType(), vegetable.getType()
equals("vegetable"));
  }

}

class Meal {

  private String type;

  public Meal(String type) {
    this.type = type;
  }

  public String getType() {
    return this.type;
  }

  // Example of factory method - different object is created depending on current context
  public static Meal valueOf(String ingredient) {
    if (ingredient.equals("banana")) {
      return new Meal("fruit");
    }
    return new Meal("vegetable");
  }
}
```

在Spring中，我们可以通过指定的工厂方法创建bean。该方法与以前代码示例中看到的valueOf方法全相同。它是静态的，可以采取没有或多个参数。为了更好地了解案例，让我们来看一下实例。首先定下配置：

```
<bean id="welcomerBean" class="com.mysite.Welcomer" factory-method="createWelcomer
>
    <constructor-arg ref="messagesLocator"></constructor-arg>
</bean>
```

```
<bean id="messagesLocator" class="com.mysite.MessageLocator">
    <property name="messages" value="messages_file.properties"></property>
</bean>
```

现在请关注这个bean的初始化：

```
public class Welcomer {

  private String message;

  public Welcomer(String message) {
    this.message = message;
  }

  public static Welcomer createWelcomer(MessageLocator messagesLocator) {
    Calendar cal = Calendar.getInstance();
    String msgKey = "welcome.pm";
    if (cal.get(Calendar.AM_PM) == Calendar.AM) {
      msgKey = "welcome.am";
    }
    return new Welcomer(messagesLocator.getMessageByKey(msgKey));
  }
}
```

当Spring将构造welcomerBean时，它不会通过传统的构造函数，而是通过定义的静态工厂方法creat Welcomer来实现。还要注意，这个方法接受一些参数（MessageLocator bean的实例包含所有可用消息） 标签，通常保留给传统的构造函数。

抽象工厂

最后一个，抽象的工厂设计模式，看起来类似于工厂方法。不同之处在于，我们可以将抽象工厂视为个词的工业意义上的工厂，即。作为提供所需对象的东西。工厂部件有：抽象工厂，抽象产品，产品客户。更准确地说，抽象工厂定义了构建对象的方法。抽象产品是这种结构的结果。产品是具有同样构的具体结果。客户是要求创造产品来抽象工厂的人。

同样的，在进入Spring的细节之前，我们将首先通过示例Java代码说明这个概念：

```
public class FactoryTest {

  // Test method which is the client
  @Test
  public void test() {
    Kitchen factory = new KitchenFactory();
    KitchenMeal meal = factory.getMeal("P.1");
    KitchenMeal dessert = factory.getDessert("I.1");
    assertTrue("Meal's name should be 'protein meal' and was '"+meal.getName()+"'", meal.ge
Name().equals("protein meal"));
    assertTrue("Dessert's name should be 'ice-cream' and was '"+dessert.getName()+"'", desser
.getName().equals("ice-cream"));
  }

}
```

```
// abstract factory
abstract class Kitchen {
  public abstract KitchenMeal getMeal(String preferency);
  public abstract KitchenMeal getDessert(String preferency);
}

// concrete factory
class KitchenFactory extends Kitchen {
  @Override
  public KitchenMeal getMeal(String preferency) {
    if (preferency.equals("F.1")) {
      return new FastFoodMeal();
    } else if (preferency.equals("P.1")) {
      return new ProteinMeal();
    }
    return new VegetarianMeal();
  }

  @Override
  public KitchenMeal getDessert(String preferency) {
    if (preferency.equals("I.1")) {
      return new IceCreamMeal();
    }
    return null;
  }
}

// abstract product
abstract class KitchenMeal {
  public abstract String getName();
}

// concrete products
class ProteinMeal extends KitchenMeal {
  @Override
  public String getName() {
    return "protein meal";
  }
}

class VegetarianMeal extends KitchenMeal {
  @Override
  public String getName() {
    return "vegetarian meal";
  }
}

class FastFoodMeal extends KitchenMeal {
  @Override
  public String getName() {
    return "fast-food meal";
  }
}
```

```
class IceCreamMeal extends KitchenMeal {
  @Override
  public String getName() {
    return "ice-cream";
  }
}
```

我们可以在这个例子中看到，抽象工厂封装了对象的创建。对象创建可以使用与经典构造函数一样使的工厂方法模式。在Spring中，工厂的例子是org.springframework.beans.factory.BeanFactory。过它的实现，我们可以从Spring的容器访问bean。根据采用的策略，getBean方法可以返回已创建对象（共享实例，单例作用域）或初始化新的对象（原型作用域）。在BeanFactory的实现中，我们以区分：ClassPathXmlApplicationContext，XmlWebApplicationContext，StaticWebApplicatio Context，StaticPortletApplicationContext，GenericApplicationContext，StaticApplicationCon ext。

```
RunWith(SpringJUnit4ClassRunner.class)

@ContextConfiguration(locations={"file:test-context.xml"})
public class TestProduct {

  @Autowired
  private BeanFactory factory;

  @Test
  public void test() {
    System.out.println("Concrete factory is: "+factory.getClass());
    assertTrue("Factory can't be null", factory != null);
    ShoppingCart cart = (ShoppingCart) factory.getBean("shoppingCart");
    assertTrue("Shopping cart object can't be null", cart != null);
    System.out.println("Found shopping cart bean:"+cart.getClass());
  }
}
```

在这种情况下，抽象工厂由BeanFactory接口表示。具体工厂是在第一个System.out中打印的，是org springframework.beans.factory.support.DefaultListableBeanFactory的实例。它的抽象产物是一对象。在我们的例子中，具体的产品就是被强转为ShoppingCart实例的抽象产品（Object）。

第一篇文章介绍了通过设计模式来正确组织的我们实现良好的编程风格。在这里，我们可以看到在Spr ng框架中使用解释器，构建器，工厂方法和工厂。第一个是帮助解释以SpEL表达的文本。三个最后模式属于创建设计模式的三剑客，它们在Spring中的主要目的是简化对象的创建。他们通过分解复杂象（构建器）的初始化或通过集中在公共点的初始化来做到对象的创建(要不然怎么叫工厂呢，必须有用点的)。

本文作者：知秋

原文链接：http://blog.didispace.com/spring-design-partern/