58 | 模板模式（上）：剖析模板模式在JDK、Servlet、JUnit等中的应用

王争 2020-03-16

00:00 08:40
讲述：冯永吉 大小：7.95M

上两节课我们学习了第一个行为型设计模式，观察者模式。针对不同的应用场景，我们讲解了不同的实现方式，有同步阻塞、异步非阻塞的实现方式，也有进程内、进程间的实现方式。除此之外，我还带你手把手实现了一个简单的 EventBus 框架。

今天，我们再学习另外一种行为型设计模式，模板模式。我们多次强调，绝大部分设计模式的原理和实现，都非常简单，难的是掌握应用场景，搞清楚能解决什么问题。模板模式也不例外。模板模式主要是用来解决复用和扩展两个问题。我们今天会结合 Java Servlet、JUnit TestCase、Java InputStream、Java AbstractList 四个例子来具体讲解这两个作用。

话不多说，让我们正式开始今天的学习吧！

模板模式的原理与实现

模板模式，全称是模板方法设计模式，英文是 Template Method Design Pattern。在 GoF 的《设计模式》一书中，它是这么定义的：

> Define the skeleton of an algorithm in an operation, deferring some steps to subclasses. Template Method lets subclasses redefine certain steps of an algorithm without changing the algorithm’ s structure.

翻译成中文就是：模板方法模式在一个方法中定义一个算法骨架，并将某些步骤推迟到子类中实现。模板方法模式可以让子类在不改变算法整体结构的情况下，重新定义算法中的某些步骤。

这里的“算法”，我们可以理解为广义上的“业务逻辑”，并不特指数据结构和算法中的“算法”。这里的算法骨架就是“模板”，包含算法骨架的方法就是“模板方法”，这也是模板方法模式名字的由来。

原理很简单，代码实现就更加简单，我写了一个示例代码，如下所示。templateMethod() 函数定义为 final，是为了避免子类重写它。method1() 和 method2() 定义为 abstract，是为了强迫子类去实现。不过，这些都不是必须的，在实际的项目开发中，模板模式的代码实现比较灵活，待会儿讲到应用场景的时候，我们会有具体的体现。

```java
public abstract class AbstractClass {
  public final void templateMethod() {
    //...
    method1();
    //...
    method2();
    //...
  }

  protected abstract void method1();
  protected abstract void method2();
}

public class ConcreteClass1 extends AbstractClass {
  @Override
  protected void method1() {
    //...
  }

  @Override
  protected void method2() {
    //...
  }
}

public class ConcreteClass2 extends AbstractClass {
  @Override
  protected void method1() {
    //...
  }

  @Override
  protected void method2() {
    //...
  }
}

AbstractClass demo = ConcreteClass1();
demo.templateMethod();
```


模板模式作用一：复用

开篇的时候，我们讲到模板模式有两大作用：复用和扩展。我们先来看它的第一个作用：复用。

模板模式把一个算法中不变的流程抽象到父类的模板方法 templateMethod() 中，将可变的部分 method1()、method2() 留给子类 ContreteClass1 和 ContreteClass2 来实现。所有的子类都可以复用父类中模板方法定义的流程代码。我们通过两个小例子来更直观地体会一下。

1.Java InputStream

Java IO 类库中，有很多类的设计用到了模板模式，比如 InputStream、OutputStream、Reader、Writer。我们拿 InputStream 来举例说明一下。

我把 InputStream 部分相关代码贴在了下面。在代码中，read() 函数是一个模板方法，定义了读取数据的整个流程，并且暴露了一个可以由子类来定制的抽象方法。不过这个方法也被命名为了 read()，只是参数跟模板方法不同。

```java
public abstract class InputStream implements Closeable {
  //...省略其他代码...

  public int read(byte b[], int off, int len) throws IOException {
    if (b == null) {
      throw new NullPointerException();
    } else if (off < 0 || len < 0 || len > b.length - off) {
      throw new IndexOutOfBoundsException();
    } else if (len == 0) {
      return 0;
    }

    int c = read();
    if (c == -1) {
      return -1;
    }
    b[off] = (byte)c;

    int i = 1;
    try {
      for (; i < len ; i++) {
        c = read();
        if (c == -1) {
          break;
        }
        b[off + i] = (byte)c;
      }
    } catch (IOException ee) {
    }
    return i;
  }

  public abstract int read() throws IOException;
}

public class ByteArrayInputStream extends InputStream {
  //...省略其他代码...

  @Override
  public synchronized int read() {
    return (pos < count) ? (buf[pos++] & 0xff) : -1;
  }
}
```

2.Java AbstractList

在 Java AbstractList 类中，addAll() 函数可以看作模板方法，add() 是子类需要重写的方法，尽管没有声明为 abstract 的，但函数实现直接抛出了 UnsupportedOperationException 异常。前提是，如果子类不重写是不能使用的。

```java
public boolean addAll(int index, Collection<? extends E> c) {
    rangeCheckForAdd(index);
    boolean modified = false;
    for (E e : c) {
        add(index++, e);
        modified = true;
    }
    return modified;
}

public void add(int index, E element) {
    throw new UnsupportedOperationException();
}
```

模板模式作用二：扩展

模板模式的第二大作用的是扩展。这里所说的扩展，并不是指代码的扩展性，而是指框架的扩展性，有点类似我们之前讲到的控制反转，你可以结合第 19 节来一块理解。基于这个作用，模板模式常用在框架的开发中，让框架用户可以在不修改框架源码的情况下，定制化框架的功能。我们通过 Junit TestCase、Java Servlet 两个例子来解释一下。

1.Java Servlet

对于 Java Web 项目开发来说，常用的开发框架是 SpringMVC。利用它，我们只需要关注业务代码的编写，底层的原理几乎不会涉及。但是，如果我们抛开这些高级框架来开发 Web 项目，必然会用到 Servlet。实际上，使用比较底层的 Servlet 来开发 Web 项目也不难。我们只需要定义一个继承 HttpServlet 的类，并且重写其中的 doGet() 或 doPost() 方法，来分别处理 get 和 post 请求。具体的代码示例如下所示：

```java
public class HelloServlet extends HttpServlet {
  @Override
  protected void doGet(HttpServletRequest req, HttpServletResponse resp) throw
    this.doPost(req, resp);
  }

  @Override
  protected void doPost(HttpServletRequest req, HttpServletResponse resp) throw
    resp.getWriter().write("Hello World.");
  }
}
```

除此之外，我们还需要在配置文件 web.xml 中做如下配置。Tomcat、Jetty 等 Servlet 容器在启动的时候，会自动加载这个配置文件中的 URL 和 Servlet 之间的映射关系。

```xml
<servlet>
    <servlet-name>HelloServlet</servlet-name>
    <servlet-class>com.xzg.cd.HelloServlet</servlet-class>
</servlet>

<servlet-mapping>
    <servlet-name>HelloServlet</servlet-name>
    <url-pattern>/hello</url-pattern>
</servlet-mapping>
```

当我们在浏览器中输入网址（比如，http://127.0.0.1:8080/hello ）的时候，Servlet 容器会接收到相应的请求，并且根据 URL 和 Servlet 之间的映射关系，找到相应的 Servlet（HelloServlet），然后执行它的 service() 方法。service() 方法定义在父类 HttpServlet 中，它会调用 doGet() 或 doPost() 方法，然后输出数据（“Hello world”）到网页。

我们现在来看，HttpServlet 的 service() 函数长什么样子。

```java
public void service(ServletRequest req, ServletResponse res)
    throws ServletException, IOException
{
    HttpServletRequest  request;
    HttpServletResponse response;
    if (!(req instanceof HttpServletRequest &&
            res instanceof HttpServletResponse)) {
        throw new ServletException("non-HTTP request or response");
    }
    request = (HttpServletRequest) req;
    response = (HttpServletResponse) res;
    service(request, response);
}

protected void service(HttpServletRequest req, HttpServletResponse resp)
    throws ServletException, IOException
{
    String method = req.getMethod();
    if (method.equals(METHOD_GET)) {
        long lastModified = getLastModified(req);
        if (lastModified == -1) {
            // servlet doesn't support if-modified-since, no reason
            // to go through further expensive logic
            doGet(req, resp);
        } else {
            long ifModifiedSince = req.getDateHeader(HEADER_IFMODSINCE);
            if (ifModifiedSince < lastModified) {
                // If the servlet mod time is later, call doGet()
                // Round down to the nearest second for a proper compare
                // A ifModifiedSince of -1 will always be less
                maybeSetLastModified(resp, lastModified);
                doGet(req, resp);
            } else {
                resp.setStatus(HttpServletResponse.SC_NOT_MODIFIED);
            }
        }
    } else if (method.equals(METHOD_HEAD)) {
        long lastModified = getLastModified(req);
        maybeSetLastModified(resp, lastModified);
        doHead(req, resp);
    } else if (method.equals(METHOD_POST)) {
        doPost(req, resp);
    } else if (method.equals(METHOD_PUT)) {
        doPut(req, resp);
    } else if (method.equals(METHOD_DELETE)) {
        doDelete(req, resp);
    } else if (method.equals(METHOD_OPTIONS)) {
        doOptions(req,resp);
    } else if (method.equals(METHOD_TRACE)) {
        doTrace(req,resp);
    } else {
        String errMsg = lStrings.getString("http.method_not_implemented");
        Object[] errArgs = new Object[1];
        errArgs[0] = method;
        errMsg = MessageFormat.format(errMsg, errArgs);
        resp.sendError(HttpServletResponse.SC_NOT_IMPLEMENTED, errMsg);
    }
}
```

从上面的代码中我们可以看出，HttpServlet 的 service() 方法就是一个模板方法，它实现了整个 HTTP 请求的执行流程，doGet()、doPost() 是模板中可以由子类来定制的部分。实际上，这就相当于 Servlet 框架提供了一个扩展点（doGet()、doPost() 方法），让框架用户在不用修改 Servlet 框架源码的情况下，将业务代码通过扩展点镶嵌到框架中执行。

2.JUnit TestCase

跟 Java Servlet 类似，JUnit 框架也通过模板模式提供了一些功能扩展点（setUp()、tearDown() 等），让框架用户可以在这些扩展点上扩展功能。

在使用 JUnit 测试框架来编写单元测试的时候，我们编写的测试类都要继承框架提供的 TestCase 类。在 TestCase 类中，runBare() 函数是模板方法，它定义了执行测试用例的整体流程：先执行 setUp() 做些准备工作，然后执行 runTest() 运行真正的测试代码，最后执行 tearDown() 做扫尾工作。

TestCase 类的具体代码如下所示。尽管 setUp()、tearDown() 并不是抽象函数，还提供了默认的实现，不强制子类去重新实现，但□这部分也是可以在子类中定制的，所以也符合模板模式的定义。

```java
public abstract class TestCase extends Assert implements Test {
  public void runBare() throws Throwable {
    Throwable exception = null;
    setUp();
    try {
      runTest();
    } catch (Throwable running) {
      exception = running;
    } finally {
      try {
        tearDown();
      } catch (Throwable tearingDown) {
        if (exception == null) exception = tearingDown;
      }
    }
    if (exception != null) throw exception;
  }

  /**
  * Sets up the fixture, for example, open a network connection.
  * This method is called before a test is executed.
  */
  protected void setUp() throws Exception {
  }

  /**
  * Tears down the fixture, for example, close a network connection.
  * This method is called after a test is executed.
  */
  protected void tearDown() throws Exception {
  }
}
```

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。

模板方法模式在一个方法中定义一个算法骨架，并将某些步骤推迟到子类中实现。模板方法模式可以让子类在不改变算法整体结构的情况下，重新定义算法中的某些步骤。这里的“算法”，我们可以理解为广义上的“业务逻辑”，并不特指数据结构和算法中的“算法”。这里的算法骨架就是“模板”，包含算法骨架的方法就是“模板方法”，这也是模板方法模式名字的由来。

在模板模式经典的实现中，模板方法定义为 final，可以避免被子类重写。需要子类重写的方法定义为 abstract，可以强迫子类去实现。不过，在实际项目开发中，模板模式的实现比较灵活，以上两点都不是必须的。

模板模式有两大作用：复用和扩展。其中，复用指的是，所有的子类可以复用父类中提供的模板方法的代码。扩展指的是，框架通过模板模式提供功能扩展点，让框架用户可以在不修改框架源码的情况下，基于扩展点定制化框架的功能。

课堂讨论

假设一个框架中的某个类暴露了两个模板方法，并且定义了一堆供模板方法调用的抽象方法，代码示例如下所示。在项目开发中，即便我们只用到这个类的其中一个模板方法，我们还是要在子类中把所有的抽象方法都实现一遍，这相当于无效劳动，有没有其他方式来解决这个问题呢？

```java
public abstract class AbstractClass {
  public final void templateMethod1() {
    //...
    method1();
    //...
    method2();
    //...
  }

  public final void templateMethod2() {
    //...
    method3();
    //...
    method4();
    //...
  }

  protected abstract void method1();
  protected abstract void method2();
  protected abstract void method3();
  protected abstract void method4();
}
```

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。