

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

王争 · 设计模式之美



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。

 复制代码

```
1 public abstract class AbstractClass {
2     public final void templateMethod() {
3         //...
4         method1();
5         //...
6         method2();
7         //...
8     }
9
10    protected abstract void method1();
11    protected abstract void method2();
12 }
13
14 public class ConcreteClass1 extends AbstractClass {
15     @Override
16     protected void method1() {
17         //...
```

```

18     }
19
20     @Override
21     protected void method2() {
22         //...
23     }
24 }
25
26 public class ConcreteClass2 extends AbstractClass {
27     @Override
28     protected void method1() {
29         //...
30     }
31
32     @Override
33     protected void method2() {
34         //...
35     }
36 }
37
38 AbstractClass demo = ConcreteClass1();
39 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，将可变的部门 `method1()`、`method2()` 留给子类 `ConcreteClass1` 和 `ConcreteClass2` 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。

```
1 public abstract class InputStream implements Closeable {
2     //...省略其他代码...
3
4     public int read(byte b[], int off, int len) throws IOException {
5         if (b == null) {
6             throw new NullPointerException();
7         } else if (off < 0 || len < 0 || len > b.length - off) {
8             throw new IndexOutOfBoundsException();
9         } else if (len == 0) {
10            return 0;
11        }
12
13        int c = read();
14        if (c == -1) {
15            return -1;
16        }
17        b[off] = (byte)c;
18
19        int i = 1;
20        try {
21            for (; i < len ; i++) {
22                c = read();
23                if (c == -1) {
24                    break;
25                }
26                b[off + i] = (byte)c;
27            }
28        } catch (IOException ee) {
29        }
30        return i;
31    }
32
33    public abstract int read() throws IOException;
34 }
35
36 public class ByteArrayInputStream extends InputStream {
37     //...省略其他代码...
38
39     @Override
40     public synchronized int read() {
41         return (pos < count) ? (buf[pos++] & 0xff) : -1;
42     }
43 }
```

2.Java ArrayList

在 Java ArrayList 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。

 复制代码

```
1 public boolean addAll(int index, Collection<? extends E> c) {
2     rangeCheckForAdd(index);
3     boolean modified = false;
4     for (E e : c) {
5         add(index++, e);
6         modified = true;
7     }
8     return modified;
9 }
10
11 public void add(int index, E element) {
12     throw new UnsupportedOperationException();
13 }
```

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：

```
1 public class HelloServlet extends HttpServlet {
2     @Override
3     protected void doGet(HttpServletRequest req, HttpServletResponse resp) throws S
4         this.doPost(req, resp);
5     }
6
7     @Override
8     protected void doPost(HttpServletRequest req, HttpServletResponse resp) throws
9         resp.getWriter().write("Hello World.");
10    }
11 }
```

复制代码

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。

```
1 <servlet>
2     <servlet-name>HelloServlet</servlet-name>
3     <servlet-class>com.xzg.cd.HelloServlet</servlet-class>
4 </servlet>
5
6 <servlet-mapping>
7     <servlet-name>HelloServlet</servlet-name>
8     <url-pattern>/hello</url-pattern>
9 </servlet-mapping>
```

复制代码

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。

```
1 public void service(ServletRequest req, ServletResponse res)
2     throws ServletException, IOException
3 {
4     HttpServletRequest request;
```

复制代码

```

5     HttpServletResponse response;
6     if (!(req instanceof HttpServletRequest &&
7         res instanceof HttpServletResponse)) {
8         throw new ServletException("non-HTTP request or response");
9     }
10    request = (HttpServletRequest) req;
11    response = (HttpServletResponse) res;
12    service(request, response);
13 }
14
15 protected void service(HttpServletRequest req, HttpServletResponse resp)
16     throws ServletException, IOException
17 {
18     String method = req.getMethod();
19     if (method.equals(METHOD_GET)) {
20         long lastModified = getLastModified(req);
21         if (lastModified == -1) {
22             // servlet doesn't support if-modified-since, no reason
23             // to go through further expensive logic
24             doGet(req, resp);
25         } else {
26             long ifModifiedSince = req.getDateHeader(HEADER_IFMODSINCE);
27             if (ifModifiedSince < lastModified) {
28                 // If the servlet mod time is later, call doGet()
29                 // Round down to the nearest second for a proper compare
30                 // A ifModifiedSince of -1 will always be less
31                 maybeSetLastModified(resp, lastModified);
32                 doGet(req, resp);
33             } else {
34                 resp.setStatus(HttpServletResponse.SC_NOT_MODIFIED);
35             }
36         }
37     } else if (method.equals(METHOD_HEAD)) {
38         long lastModified = getLastModified(req);
39         maybeSetLastModified(resp, lastModified);
40         doHead(req, resp);
41     } else if (method.equals(METHOD_POST)) {
42         doPost(req, resp);
43     } else if (method.equals(METHOD_PUT)) {
44         doPut(req, resp);
45     } else if (method.equals(METHOD_DELETE)) {
46         doDelete(req, resp);
47     } else if (method.equals(METHOD_OPTIONS)) {
48         doOptions(req, resp);
49     } else if (method.equals(METHOD_TRACE)) {
50         doTrace(req, resp);
51     } else {
52         String errMsg = lStrings.getString("http.method_not_implemented");
53         Object[] errArgs = new Object[1];

```



```
54         errArgs[0] = method;
55         errMsg = MessageFormat.format(errMsg, errArgs);
56         resp.sendError(HttpServletResponse.SC_NOT_IMPLEMENTED, errMsg);
57     }
58 }
```


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。

 复制代码

```
1 public abstract class TestCase extends Assert implements Test {
2     public void runBare() throws Throwable {
3         Throwable exception = null;
4         setUp();
5         try {
6             runTest();
7         } catch (Throwable running) {
8             exception = running;
9         } finally {
10            try {
11                tearDown();
12            } catch (Throwable tearingDown) {
```



```
13         if (exception == null) exception = tearingDown;
14     }
15 }
16     if (exception != null) throw exception;
17 }
18
19 /**
20  * Sets up the fixture, for example, open a network connection.
21  * This method is called before a test is executed.
22  */
23     protected void setUp() throws Exception {
24     }
25
26 /**
27  * Tears down the fixture, for example, close a network connection.
28  * This method is called after a test is executed.
29  */
30     protected void tearDown() throws Exception {
31     }
32 }
```

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？

 复制代码

```
1 public abstract class AbstractClass {
2     public final void templateMethod1() {
3         //...
4         method1();
5         //...
6         method2();
7         //...
8     }
9
10    public final void templateMethod2() {
11        //...
12        method3();
13        //...
14        method4();
15        //...
16    }
17
18    protected abstract void method1();
19    protected abstract void method2();
20    protected abstract void method3();
21    protected abstract void method4();
22 }
```

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

模板模式在Java开发中的应用是本文的重点。通过分析Java Servlet、JUnit TestCase、Java InputStream、Java AbstractList等实际例子，深入讲解了模板模式的原理与实现。模板模式的作用主要包括复用和扩展两个方面。在复用方面，模板模式将不变的流程抽象到父类的模板方法中，可变的部分留给子类实现，实现了代码的复用。在扩展方面，模板模式提供了扩展点，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。通过具体的例子如Java Servlet和Java InputStream的分析，展示了模板模式在框架开发中的应用。文章通过清晰的代码示例和实际应用场景，帮助读者深入理解模板模式的原理和实现，以及在Java开发中的实际应用。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，实现了算法的复用和扩展。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。

© 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 (112)

最新 精选



Geek_cead38

2020-11-22

我觉得即便是使用装饰器还是直接重写method1-4，对于需要子类重写的方法要么抛不支持异常，要么抽象，不然子类察觉不到必须重写，导致模板函数的业务出错

作者回复: 🤔🤔🤔🤔🤔🤔

共 3 条评论 >



3



Laughing

2020-11-24

结尾问题有两种解决方法：

1. 写适配器来转化
2. 提供默认实现不要求强制实现

作者回复: 🤔🤔🤔🤔🤔🤔



攻城拔寨

2020-03-17

文末的问题，在 spring 生命周期中，InstantiationAwareBeanPostProcessorAdapter 就是解决这个问题的。

写个适配器，把所有抽象方法默认实现一下，子类继承这个 adapter 就行了。

共 16 条评论 >



230



Eclipse

2020-03-16

可以借鉴AbstractList的addAll实现。提供默认的方法method1...method4方法，每个方法直接抛出异常，使用模板方法的时候强制重写用到的method方法，用不到的method不用重写。

共 10 条评论 >

👍 107



Rayjun

2020-03-16

如果两个模版方法没有耦合，可以拆分成两个类，如果不能拆分，那就为每个方法提供默认实现



👍 73



最好的狗焕啊

2020-03-17

争哥，一年前就很崇拜你了，但是现在很迷茫，三年的开发经验了，一直在小公司，做的项目最多的数据量也只是十几万的用户，平常下班每天都会坚持学习两个小时，已经坚持一年半了，看了数据结构和算法，还有认真刷过题，看了网络协议，也看了框架方面的书等等，也认真做了笔记，然后想投递独角兽公司，但是简历都不通过，理由是学历和项目都没有亮点，我是本科毕业，看了网上的一些阿里或者百度这样的公司的面试题，发现自己也会，但是投递的简历都不通过，真的很迷茫，不知道这样的坚持有没有用，现在想回到老家一个二线城市，做着一份养老的工作

共 43 条评论 >

👍 52



tt

2020-03-17

参考装饰器模式那一课中JAVA IO类库中的做法，引入一个中间父类，实现所有的抽象方法，然后再让业务类去继承这个中间父类。



👍 26



每天晒白牙

2020-03-16

提供一个 Base 类，实现 method1 到 method4 的所有抽象方法，然后子类继承 Base 类，一般可以直接复用 Base 类中的 method1 到 method4 方法，如果需要重写，直接重写该方法就好。这样就能省去所有子类实现所有抽象方法

继承抽象方法的基类 Base

```
public class Base extends AbstractClass {
```

```

@Override
protected void method1() {
    System.out.println("1");
}

@Override
protected void method2() {
    System.out.println("2");
}

@Override
protected void method3() {
    System.out.println("3");
}

@Override
protected void method4() {
    System.out.println("4");
}
}

```

子类 A 需要重写 method1 方法

```

public class SubA extends Base {

    // 只需要重写 method1
    @Override
    public void method1() {
        System.out.println("重写 method1");
    }

    public static void main(String[] args) {
        Base A = new SubA();
        A.templateMethod1();
    }
}

```

输出结果为

重写 method1

2



👍 26



小兵

2020-03-16

父类中不用抽象方法，提供一个空的实现，子类根据需要重写。

共 3 条评论 >

👍 17



下雨天

2020-03-16

课后思考：

一. 能修改框架代码情况：

定义一个父类，给不需要调用的抽象方法一个默认实现，子类继承该父类。

二. 如果可以修改框架代码的情况下：

1.templateMethod1与templateMethod2相关：可以将不需要调用的方法修改成protected并提供默认空实现。

2.templateMethod1与templateMethod2不相关：接口隔离原则，考虑将AbstractClass拆分成两个类分别定义两个方法。



👍 10