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

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。

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

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

public class ContreteClass1 extends AbstractClass {
  @Override
  protected void method1() {
    //...
  }
```

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

public class ContreteClass2 extends AbstractClass {
  @Override
  protected void method1() {
    //...
  }

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

AbstractClass demo = ContreteClass1();
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。

```
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。

```
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：

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

  @Override
  protected void doPost(HttpServletRequest req, HttpServletResponse resp) throws ServletException, IOExcept
    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。

```
<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。

```
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 {
        //
        // Note that this means NO servlet supports whatever
        // method was requested, anywhere on this server.
        //

        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。

```
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？

```
public 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。

精选留言：

- rayjun 2020-03-16 07:35:54
  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 [7赞]

- 小兵 2020-03-16 07:15:10
  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。 [5赞]

- 付昱霖 2020-03-16 09:13:29
  使用外观模式，用一个新类再次包装，只暴露需要的接口。 [1赞]

- 自来也 2020-03-16 07:50:33
  Es框架里，abstractrunable是属于包装者还是模板。感觉更像包装者。不管啥了，总之觉得这样挺好用的。父类public就好了，就能解决没必要强制重写了。 [1赞]

- 刘大明 2020-03-16 07:31:25
  如果其他的类不考虑复用的话，可以将这些抽取成一个基类，就是两个抽象类。分别给不需要的方法定义空实现。 [1赞]

- 宁锟 2020-03-16 07:08:06
  定义两个抽象类，继承模板类，分别给不需要的方法定义空实现 [1赞]

- LJK 2020-03-16 02:25:57
  课后作业的思考：对于必须要子类实现的方法定义为抽象方法或throw Exception，对于变动比较少但是同时也不想失去扩展性的方法添加默认实现，调用时优先获取用户自定义方法，获取不到的情况下使用默认方法 [1赞]

- Sinclairs 2020-03-16 01:45:28
  如果项目中多次用到这个类的话, 可以单独实现一个基类来继承这个模版类, 将不需要的扩展方法进行默认实现.
  项目开发中直接使用基类方法就好. [1赞]

- rookie 2020-03-16 14:12:42
  根据问题描述，有两个templateMethod1()和templateMethod2()模板方法，其中实现调用的方法并没有并集，可以拆分成两个类。

- 每天晒白牙 2020-03-16 13:16:54
  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
```

- yan华建 2020-03-16 12:45:05
  课堂讨论：
  方法一：接口空实现。
  方法二：定义新的抽象类，不需要使用的接口采用空实现的方式，拓展类基于该抽象类进行实现。

- 小晏子 2020-03-16 12:41:45
  首先我认为不应该拆开成两个类，因为如果子类需要实现所有的模版方法，因为java不支持多继承，拆成两个类就没法搞了。简单一点的办法就是把所有要实现的方法变成非抽象的，内部抛出异常, 使其直接使用就报错，只能继承。

- iLeGeND 2020-03-16 12:31:09
  缺省适配器

- 正在减肥的胖籽。 2020-03-16 11:24:09
  拆分成2个模板方法。

- Frank 2020-03-16 11:12:07
  打卡 今日学习模版模式，收获：在一个方法中定义了一个算法骨架，并将某些步骤推迟到子类中实现。在我们的项目中也用到了这种模式，比如我们需要对接多个BU的系统，他们之间是通过消息来进行通信，我们定义好一个模板（处理流程），这样各个业务系统的对接时保证主要的流程不变，在预留的扩展点

上就行代码编写，从而完成业务。
对于课堂讨论我的思考是：引入一个中间类来实现抽象方法，空实现。然后在具体的子类中继承这个中间类，选择性的复写需要的方法。

- Eden Ma 2020-03-16 10:51:19
参考JUnit TestCase给抽象方法设置默认实现.

- Heaven 2020-03-16 10:38:33
最简单的解决方案,就是对其一个个的实现默认方法,然后在默认实现中,抛出一个运行时异常,但是这种实现方式,很可能导致编写代码时候的不注意,没有在需要调用的方法中,实现原本的抽象方法从而出现异常,而且,按理来说,模板模式一般应用于业务逻辑相关的函数中,为了符合单一职责原则,尽量不要在一个类中出现两个乃至多个业务逻辑相关函数,于是,应该把多个业务逻辑函数拆分到不同的类中

- 守拙 2020-03-16 10:03:42
课堂讨论:
考虑使用Adapter Pattern解决接口隔离问题.

```
public abstract class AbsClass {

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();

}

public class AbsAdapter extends AbsClass {

@Override
protected void method1() {
throw new IllegalStateException("need subclass implementaion");
}

@Override
protected void method2() {
throw new IllegalStateException("need subclass implementaion");
}
```

```
@Override
protected void method3() {
throw new IllegalStateException("need subclass implementaion");
}

@Override
protected void method4() {
throw new IllegalStateException("need subclass implementaion");
}
}

/**
* ConcreteClass1 只用到templateMethod1()*/
public class ConcreteClass1 extends AbsAdapter {

public ConcreteClass1() {
}

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

public class TestTemplate {

public static void main(String[] args) {
AbsClass instance1 = new ConcreteClass1();
AbsClass instance2 = new ConcreteClass2();
instance1.templateMethod1();
instance2.templateMethod2();
}
}
@Override
protected void method2() {
System.out.println("method2 exec");
}
}
```

- Jeff.Smile 2020-03-16 09:49:37

要点总结：
①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
② 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。
③ 模板方法模式的两大特性：可复用和可扩展
思考题：
抽象方法改成非抽象的并提供默认的实现。

- Jackey 2020-03-16 09:41:33
  提供默认实现，子类只重写自己需要的方法，其他走父类的默认实现是不是就可以了