《JAVA与模式》之模板方法模式

在阎宏博士的《JAVA与模式》一书中开头是这样描述模板方法（Template Method）模式的：

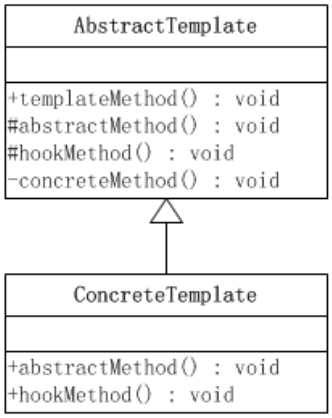
模板方法模式是类的行为模式。准备一个抽象类，将部分逻辑以具体方法以及具体构造函数的形式实现，然后声明一些抽象方法来迫使子类实现剩余的逻辑。不同的子类可以以不同的方式实现这些抽象方法，从而对剩余的逻辑有不同的实现。这就是模板方法模式的用意。

模板方法模式的结构

模板方法模式是所有模式中最为常见的几个模式之一，是基于继承的代码复用的基本技术。

模板方法模式需要开发抽象类和具体子类的设计师之间的协作。一个设计师负责给出一个算法的轮廓和骨架，另一些设计师则负责给出这个算法的各个逻辑步骤。代表这些具体逻辑步骤的方法称做基本方法(primitive method)；而将这些基本方法汇总起来的方法叫做模板方法(template method)，这个设计模式的名字就是从此而来。

模板方法所代表的行为称为顶级行为，其逻辑称为顶级逻辑。模板方法模式的静态结构图如下所示：



这里涉及到两个角色：

抽象模板(Abstract Template)角色有如下责任：

- 定义了一个或多个抽象操作，以便让子类实现。这些抽象操作叫做基本操作，它们是一个顶级逻辑的组成步骤。
- 定义并实现了一个模板方法。这个模板方法一般是一个具体方法，它给出了一个顶级逻辑的骨架，而逻辑的组成步骤在相应的抽象操作中，推迟到子类实现。顶级逻辑也有可能调用一些具体方法。

具体模板(Concrete Template)角色又如下责任：

- 实现父类所定义的一个或多个抽象方法，它们是一个顶级逻辑的组成步骤。
- 每一个抽象模板角色都可以有任意多个具体模板角色与之对应，而每一个具体模板角色都可以给出这些抽象方法（也就是顶级逻辑的组成步骤）的不同实现，从而使得顶级逻辑的实现各不相同。

源代码

抽象模板角色类，abstractMethod()、hookMethod()等基本方法是顶级逻辑的组成步骤，这个顶级逻辑由templateMethod()方法代表。

```
public abstract class AbstractTemplate {
    /**
     * 模板方法
     */
    public void templateMethod(){
        //调用基本方法
        abstractMethod();
        hookMethod();
        concreteMethod();
    }
    /**
     * 基本方法的声明（由子类实现）
     */
    protected abstract void abstractMethod();
    /**
     * 基本方法(空方法)
     */
    protected void hookMethod(){}
    /**
     * 基本方法（已经实现）
```

```
     */
    private final void concreteMethod(){
        //业务相关的代码
    }
}
```

具体模板角色类，实现了父类所声明的基本方法，abstractMethod()方法所代表的就是强制子类实现的剩余逻辑，而hookMethod()方法是可选择实现的逻辑，不是必须实现的。

```
public class ConcreteTemplate extends AbstractTemplate{
    //基本方法的实现
    @Override
    public void abstractMethod() {
        //业务相关的代码
    }
    //重写父类的方法
    @Override
    public void hookMethod() {
        //业务相关的代码
    }
}
```

模板模式的关键是：**子类可以置换掉父类的可变部分，但是子类却不可以改变模板方法所代表的顶级逻辑。**

每当定义一个新的子类时，不要按照控制流程的思路去想，而应当按照“责任”的思路去想。换言之，应当考虑哪些操作是必须置换掉的，哪些操作是可以置换掉的，以及哪些操作是不可以置换掉的。使用模板模式可以使这些责任变得清晰。

模板方法模式中的方法

模板方法中的方法可以分为两大类：模板方法和基本方法。

模板方法

一个模板方法是定义在抽象类中的，把基本操作方法组合在一起形成一个总算法或一个总行为的方法。

一个抽象类可以有任意多个模板方法，而不限于一个。每一个模板方法都可以调用任意多个具体方法。

基本方法

基本方法又可以分为三种：抽象方法(Abstract Method)、具体方法(Concrete Method)和钩子方法(Hook Method)。

- **抽象方法：**一个抽象方法由抽象类声明，由具体子类实现。在Java语言里抽象方法以abstract关键字标示。
- **具体方法：**一个具体方法由抽象类声明并实现，而子类并不实现或置换。
- **钩子方法：**一个钩子方法由抽象类声明并实现，而子类会加以扩展。通常抽象类给出的实现是一个空实现，作为方法的默认实现。

在上面的例子中，AbstractTemplate是一个抽象类，它带有三个方法。其中abstractMethod()是一个抽象方法，它由抽象类声明为抽象方法，并由子类实现；hookMethod()是一个钩子方法，它由抽象类声明并提供默认实现，并且由子类置换掉。concreteMethod()是一个具体方法，它由抽象类声明并实现。

默认钩子方法

一个钩子方法常常由抽象类给出一个空实现作为此方法的默认实现。这种空的钩子方法叫做“Do Nothing Hook”。显然，这种默认钩子方法在缺省适配模式里面已经见过了，一个缺省适配模式讲的是一个类为一个接口提供一个默认的空实现，从而使得缺省适配类的子类不必像实现接口那样必须给出所有方法的实现，因为通常一个具体类并不需要所有的方法。

命名规则

命名规则是设计师之间赖以沟通的管道之一，使用恰当的命名规则可以帮助不同设计师之间的沟通。

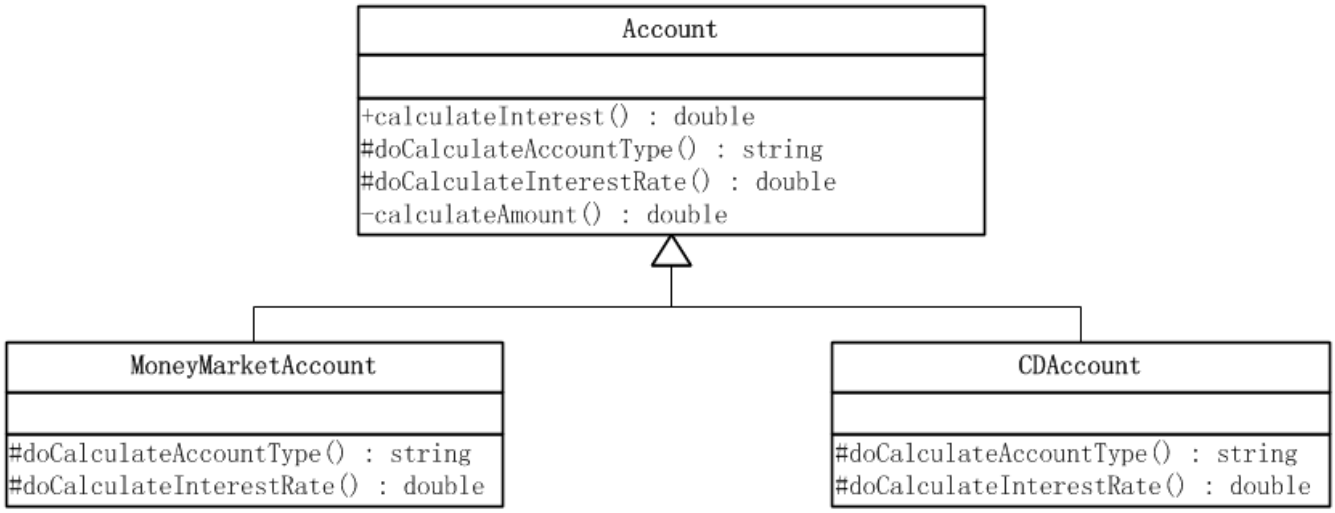
钩子方法的名字应当以do开始，这是熟悉设计模式的Java开发人员的标准做法。在上面的例子中，钩子方法hookMethod()应当以do开头；在HttpServlet类中，也遵从这一命名规则，如doGet()、doPost()等方法。

使用场景

考虑一个计算存款利息的例子。假设系统需要支持两种存款账号，即货币市场(Money Market)账号和定期存款(Certificate of Deposite)账号。这两种账号的存款利息是不同的，因此，在计算一个存户的存款利息额时，必须区分两种不同的账号类型。

这个系统的总行为应当是计算出利息，这也就决定了作为一个模板方法模式的顶级逻辑应当是利息计算。由于利息计算涉及到两个步骤：一个基本方法给出账号种类，另一个基本方法给出利息百分比。这两个基本方法构成具体逻辑，因为账号的类型不同，所以具体逻辑会有所不同。

显然，系统需要一个抽象角色给出顶级行为的实现，而将两个作为细节步骤的基本方法留给具体子类实现。由于需要考虑的账号有两种：一是货币市场账号，二是定期存款账号。系统的类结构如下图所示。



源代码

抽象模板角色类

```
public abstract class Account {
    /**
     * 模板方法，计算利息数额
     * @return    返回利息数额
     */
    public final double calculateInterest(){
        double interestRate = doCalculateInterestRate();
        String accountType = doCalculateAccountType();
        double amount = calculateAmount(accountType);
        return amount * interestRate;
    }
    /**
     * 基本方法留给子类实现
     */
    protected abstract String doCalculateAccountType();
    /**
     * 基本方法留给子类实现
     */
    protected abstract double doCalculateInterestRate();
    /**
     * 基本方法，已经实现
     */
    private double calculateAmount(String accountType){
        /**
         * 省略相关的业务逻辑
         */
        return 7243.00;
    }
}
```

具体模板角色类

```
public class MoneyMarketAccount extends Account {

    @Override
```

```
    protected String doCalculateAccountType() {

        return "Money Market";
    }

    @Override
    protected double doCalculateInterestRate() {

        return 0.045;
    }

}
```

```
public class CDAccount extends Account {

    @Override
    protected String doCalculateAccountType() {
        return "Certificate of Deposite";
    }

    @Override
    protected double doCalculateInterestRate() {
        return 0.06;
    }

}
```

客户端类

```
public class Client {

    public static void main(String[] args) {
        Account account = new MoneyMarketAccount();
        System.out.println("货币市场账号的利息数额为：" + account.calculateInterest());
        account = new CDAccount();
        System.out.println("定期账号的利息数额为：" + account.calculateInterest());
    }

}
```

模板方法模式在Servlet中的应用

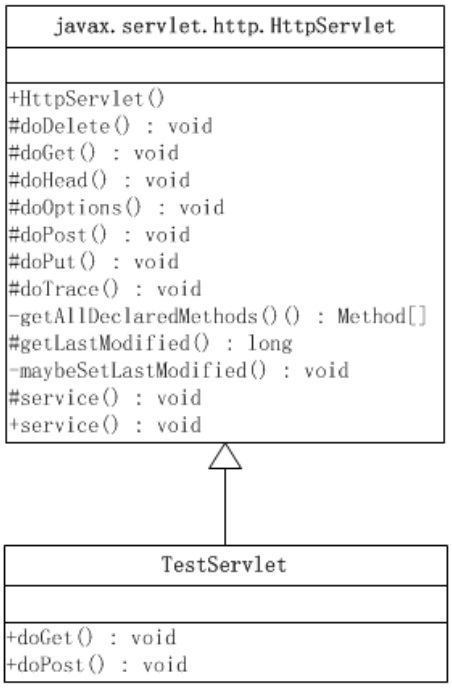
使用过Servlet的人都清楚，除了要在web.xml做相应的配置外，还需继承一个叫HttpServlet的抽象类。HttpService类提供了一个service()方法，这个方法调用七个do方法中的一个或几个，完成对客户端调用的响应。这些do方法需要由HttpServlet的具体子类提供，因此这是典型的模板方法模式。下面是service()方法的源代码：

```
    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 / 1000 * 1000)) {
                    // 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);
        }
    }
```

当然，这个service()方法也可以被子类置换掉。

下面给出一个简单的Servlet例子：



从上面的类图可以看出，TestServlet类是HttpServlet类的子类，并且置换掉了父类的两个方法：doGet()和doPost()。

```
public class TestServlet extends HttpServlet {

    public void doGet(HttpServletRequest request, HttpServletResponse response)
            throws ServletException, IOException {

        System.out.println("using the GET method");

    }

    public void doPost(HttpServletRequest request, HttpServletResponse response)
            throws ServletException, IOException {

        System.out.println("using the POST method");
    }

}
```

从上面的例子可以看出这是一个典型的模板方法模式。

HttpServlet担任抽象模板角色

模板方法：由service()方法担任。

基本方法：由doPost()、doGet()等方法担任。

TestServlet担任具体模板角色

TestServlet置换掉了父类HttpServlet中七个基本方法中的其中两个，分别是doGet()和doPost()。