概述

> 模板方法模式在一个方法中定义一个算法的骨架，而将一些步骤延迟到子类中。模板方法使得子类可以在不改变算法结构的情况下，重新定义算法中的某些步骤。

不知道在场有没有喜欢烹饪的小伙伴，做一道西红柿炒蛋我们通常需要以下步骤：

1. 打开抽油烟机
2. 生火
3. 西红柿炒蛋
4. 关火
5. 关闭抽油烟机

用代码来表示：

```
public class CookXihongshi {

    public void open(){
        System.out.println("打开抽油烟机");
    }
    public void fire(){
        System.out.println("生火");
    }
    public void doCook(){
        System.out.println("西红柿炒蛋");
    }
    public void outfire(){
        System.out.println("灭火");
    }
    public void close(){
        System.out.println("关闭抽油烟机");
    }
    public void cook(){
        this.open();
        this.fire();
        this.doCook();
        this.outfire();
        this.close();
    }
}
```

执行：

```
CookXihongshi cook = new CookXihongshi();
cook.cook();
```

如果这时候又要炒个青菜：

```
public class CookVegetable {

    public void open(){
        System.out.println("打开抽油烟机");
    }
    public void fire(){
        System.out.println("生火");
    }
    public void doCook(){
        System.out.println("炒青菜");
    }
    public void outfire(){
        System.out.println("灭火");
    }
    public void close(){
        System.out.println("关闭抽油烟机");
    }
    public void cook(){
        this.open();
        this.fire();
        this.doCook();
        this.outfire();
        this.close();
    }
}
```

炒青菜的类和上面西红柿炒蛋的类一比较，你会发现除了doCook()方法中具体的实现不一样之外，其他步骤都是完全一模一样的。身为程序猿，一看到重复代码就觉得烦。解决办法很简单，将重复代码抽出来让父类去实现，而西红柿炒蛋和炒青菜都继承它，这样就避免了重复代码：

抽象类：

```
public abstract class Cook {

    public void open(){
        System.out.println("打开抽油烟机");
    }
    public void fire(){
        System.out.println("生火");
    }
    /**
     * 期望子类去实现
     */
    public abstract void doCook();

    public void outfire(){
        System.out.println("灭火");
    }
    public void close(){
        System.out.println("关闭抽油烟机");
    }
    /**
```

```
     * 使用final关键字，防止子类重写
     */
    public final void cook(){
        this.open();
        this.fire();
        this.doCook();
        this.outfire();
        this.close();
    }
}
```

该类中有两个地方是关键，一是doCook()使用abstract修饰，让子类去实现。二是cook()方法使用final关键字修饰，防止子类重写，从而破坏了模板中规定好的流程。但这两点并不是强制要求这么做的，可以视情况而定。

接下来西红柿炒蛋和炒青菜都只要实现他们不同的那部分代码即可。

西红柿炒蛋类：

```
public class CookXihongshi extends Cook {

    @Override
    public void doCook() {
        System.out.println("西红柿炒蛋");
    }
}
```

炒青菜

```
public class CookVegetable extends Cook {

    @Override
    public void doCook() {
        System.out.println("炒青菜");
    }
}
```

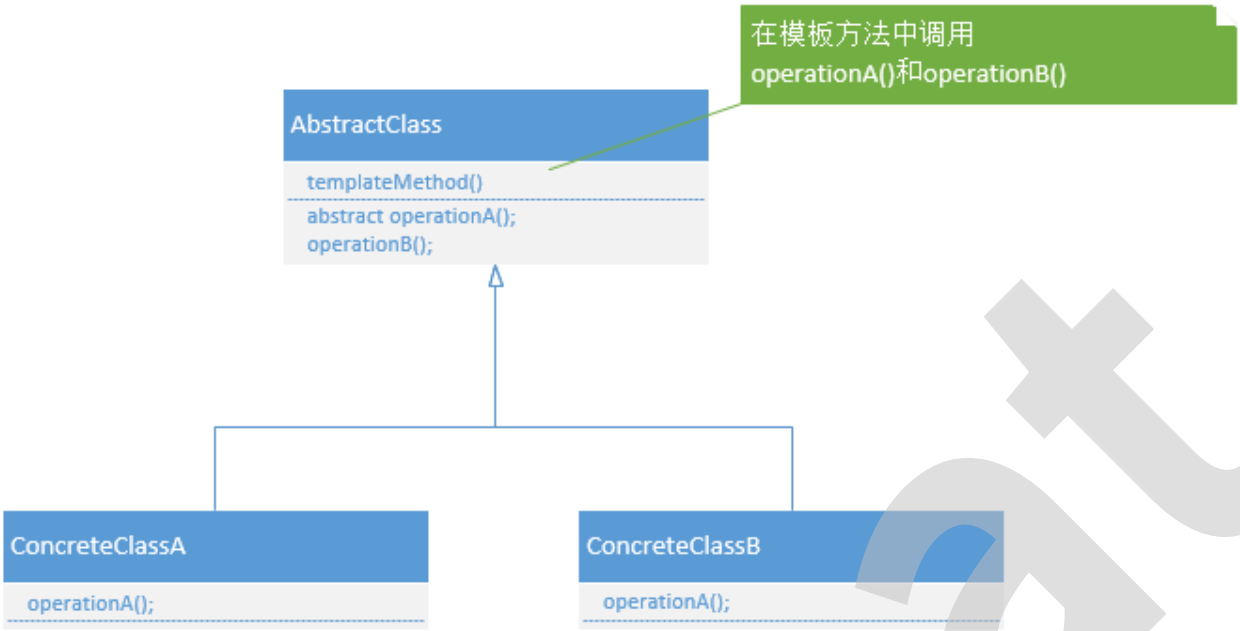
调用：

```
Cook cook = new CookXihongshi();
cook.cook();
```

看到这里，没错，你已经学会了模板方法模式。模板方法是设计模式中较好理解的一种，他的使用场景是：当有一个业务有N个步骤（模板），其中一部分步骤是永恒不变的，那么就将不变的这些步骤抽象到父类中，可能变化的步骤留给子类去实现。

在上面烹饪的例子中，`cook()`方法中调用了开启抽油烟机，生火，开炒，灭火，关闭抽油烟机等步骤。不管炒什么菜，这些步骤都是一样的，像模板一样。因此`cook()`就是我们所说的**“模板方法”**。

模板方法模式类图如下：



模板中的钩子

关于钩子的概念，这里不再阐述，相信大部分朋友对此都不陌生。新手朋友请自行度娘。

在模板方法模式中，父类可以提供一个默认为空的方法，如 `beforeCook()` ，并在模板方法中调用：

```
public abstract class Cook {

    public void open(){
        System.out.println("打开抽油烟机");
    }
    public void fire(){
        System.out.println("生火");
    }

    /**
     * 期望子类去实现
     */
    public abstract void doCook();

    public void outfire(){
        System.out.println("灭火");
    }
    public void close(){
        System.out.println("关闭抽油烟机");
    }

    public void beforeCook(){};

    /**
     * 使用final关键字，防止子类重写
     */
    public final void cook(){
```

```
        beforeCook();
        this.open();
        this.fire();
        this.doCook();
        this.outfire();
        this.close();
    }
}
```

上面代码中增加了`beforeCook()`，什么也不做，并且在模板方法`cook()`中调用它。

假如某位厨神每次在做西红柿炒蛋的之前都要焚香三日，沐浴更衣，那么子类只需要重写该方法即可：

```
public class CookXihongshi extends Cook {

    @Override
    public void beforeCook() {
        System.out.println("焚香三日，沐浴更衣");
    }

    @Override
    public void doCook() {
        System.out.println("西红柿炒蛋");
    }
}
```

这个`beforeCook()`就是一个钩子，在模板方法中我们可以提前预埋多个钩子，让子类有一定的能力影响抽象类中的算法流程。

可是突然某一天那位厨神做西红柿炒蛋之前不想焚香沐浴了咋办，难道把`beforeCook()`删掉吗？不是不可以，可是哪天又突然想沐浴焚香了咋办？再写一遍？如果想根据星期，一三五焚香沐浴，二四六七不焚香沐浴，咋整？

很简单，我们可以给钩子设置一个开关：

```
    /**
     * 钩子的开关
     * @return
     */
    protected boolean needBeforeCook(){
        return true;
    }

    /**
     * 使用final关键字，防止子类重写
     */
    public final void cook(){

        if(needBeforeCook()){
            beforeCook();
        }


        this.open();
```

```
        this.fire();
        this.doCook();
        this.outfire();
        this.close();
    }
```

默认开启钩子，想要关闭，子类需重写`needBeforeCook()`。

也可以使用变量作为开关：

```
protected boolean needBeforeCook = true;

public final void cook(){
    if(this.needBeforeCook){
        beforeCook();
    }

    this.open();
    this.fire();
    this.doCook();
    this.outfire();
    this.close();
}
```

用变量的方式相对灵活一些，子类可以根据业务动态地启用或关闭钩子。

其实看到`beforeCook`这样的命名就有很多朋友觉得眼熟，这不就是传说中的“前置操作”和“后置操作”吗？没错。前置和后置操作其实就是模板方法模式的应用。

实际案例

Servlet中的模板方法

开发web应用或api接口，我们必然要用到servlet，一般步骤是写一个类继承`HttpServlet`，然后重写doGet或者doPost方法来分别处理get请求和post请求：

```
public class Test extends HttpServlet {

    @Override
    protected void doGet(HttpServletRequest req, HttpServletResponse resp) throws
ServletException, IOException {
        System.out.println("处理Get请求");
        super.doGet(req, resp);
    }
    @Override
    protected void doPost(HttpServletRequest req, HttpServletResponse resp) throws
ServletException, IOException {
        System.out.println("处理Post请求");
        super.doPost(req, resp);
    }
}
```

其实这也就是用了模板方法，我们打开 `HttpServlet` 类的源码，发现除了doGet，doPost外还有很多类似的方法，如doDelete，doPut等。有一个共通点，它们都是在 `service()` 方法中被调用：

```
    public void service(ServletRequest req, ServletResponse res) throws ServletException,
IOException {
        HttpServletRequest request;
        HttpServletResponse response;
        try {
            request = (HttpServletRequest)req;
            response = (HttpServletResponse)res;
        } catch (ClassCastException var6) {
            throw new ServletException("non-HTTP request or response");
        }

        this.service(request, response);
    }
    protected void service(HttpServletRequest req, HttpServletResponse resp) throws
ServletException, IOException {
        String method = req.getMethod();
        long lastModified;
        if (method.equals("GET")) {
            lastModified = this.getLastModified(req);
            if (lastModified == -1L) {
                this.doGet(req, resp);
            } else {
                long ifModifiedSince;
                try {
                    ifModifiedSince = req.getDateHeader("If-Modified-Since");
                } catch (IllegalArgumentException var9) {
                    ifModifiedSince = -1L;
                }

                if (ifModifiedSince < lastModified / 1000L * 1000L) {
                    this.maybeSetLastModified(resp, lastModified);
                    this.doGet(req, resp);
                } else {
```

```
                    resp.setStatus(304);
                }
            }
        } else if (method.equals("HEAD")) {
            lastModified = this.getLastModified(req);
            this.maybeSetLastModified(resp, lastModified);
            this.doHead(req, resp);
        } else if (method.equals("POST")) {
            this.doPost(req, resp);
        } else if (method.equals("PUT")) {
            this.doPut(req, resp);
        } else if (method.equals("DELETE")) {
            this.doDelete(req, resp);
        } else if (method.equals("OPTIONS")) {
            this.doOptions(req, resp);
        } else if (method.equals("TRACE")) {
            this.doTrace(req, resp);
        } else {
            String errMsg = lStrings.getString("http.method_not_implemented");
            Object[] errArgs = new Object[]{method};
            errMsg = MessageFormat.format(errMsg, errArgs);
            resp.sendError(501, errMsg);
        }

    }
```

`service()` 具体实现只要简单地浏览一下就好，不用详细看，只需知道在service内部根据http请求方式不同分别调用doGet，doPost等方法。`service()` 就是模板方法，doGet等方法就是需要子类去实现的方法。

模板方法在开发中被广泛应用，只要看到跟Servlet这种类似的类，就很有可能用了模板方法。大家有碰到不妨点进源码看一看，有助于加深对模板方法的理解。

排序

在JAVA中，对一组对象进行排序可以使用 `Arrays.sort()` 方法，它要求对象实现 `Comparable` 接口，示例如下：

```
public class Person implements Comparable{

    private Integer age;

    public Person(Integer age){
        this.age = age;
    }

    @Override
    public int compareTo(Object o) {
        Person person = (Person)o;
        return this.age.compareTo(person.age);
    }

    @Override
    public String toString() {
```

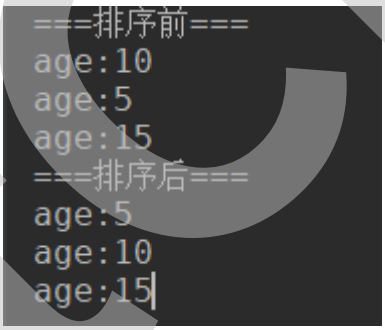
```
        return "age:" + this.age;
    }
}
```

测试：

```
Person p1 = new Person(10);
Person p2 = new Person(5);
Person p3 = new Person(15);
Person[] persons = {p1,p2,p3};

System.out.println("===排序前===");
for (int i=0; i< persons.length;i++){
    System.out.println(persons[i]);
}
//排序
Arrays.sort(persons);
System.out.println("===排序后===");
for (int i=0; i< persons.length;i++){
    System.out.println(persons[i]);
}
```

结果：



sort排序方法是如何实现的呢？其实原理很简单，其中有一段代码如下：

```
if (length < INSERTIONSORT_THRESHOLD) {
     for(int i=low; i<high; i++)
         for (int j=i; j>low &&
                  ((Comparable) dest[j-1]).compareTo(dest[j])>0; j--)
             swap(dest, j, j-1);
     return;
 }
```

说白了就是调用对象的`compareTo()`方法，如果大小相反，则用`swap()`调换一下顺序。

其实这也是模板方法的应用。之前说到模板方法是将不变的步骤抽象到父类中，可是排序这里明明没有啊，结构完全不一样，咋回事呢？

设计模式传递给我们的是编程思想，而不是固定的代码模板。

为了让所有的对象数组都支持这个排序方法，要求排序对象实现 `Comparable` 接口，这是一种基于“约定”的“继承”，而不是编译层面的“继承”。而 `Arrays.sort()` 方法其实就是模板方法。

不得不说，JAVA中的许多设计真是令人叹为观止。

总结

模板方法模式是将子类中不变的部分抽象到父类，可变的部分由子类去实现。

优点

封装不变公共代码，便于维护。可变部分的代码由子类自由决定，扩展性强

缺点

每新增一个不同的实现都需要增加一个子类，可能导致类数量变多，增加系统复杂性