

模板方法模式

一、简介

定义了一个 算法 的 骨架 , 并允许 子类 为 一个或多个 步骤 提供实现 ;
可以使 子类 在 不改变 算法结构 的前提下 , 重新定义算法的某些步骤 ;

二、模式的结构

模板方法模式包含以下主要角色:

- (1) 抽象类/抽象模板
- 抽象模板类负责给出一个算法的轮廓和骨架。它由一个模板方法和若干的基本方法构成。
- 这些方法定义如下。
1. 模板方法: 定义了算法的骨架, 按某种顺序调用其包含的基本方法。

2. 基本方法: 是整个算法中的一个步骤, 包含以下几种类型:
 - 具体方法: 在抽象类中已经实现, 在具体子类中可以继承或重写它。
 - 抽象方法: 在抽象类中声明, 交给具体子类去实现。
 - 钩子方法: 在抽象类中已经实现, 一般为默认实现的一个空方法或者用于判断的逻辑方法, 作为某个特定步骤的开关
可以通过子类重写来确定某个步骤是否需要开启
- (2) 具体子类/具体实现
- 需要实现抽象类中所定义的抽象方法和钩子方法, 它们是模板方法中的组成部分
- 所有的步骤方法都是protected修饰的, 因为我们希望具体算法的实现只有子类可以访问, 对外不开放。

三、基本模板方法的代码实现

以在力扣上刷题为例子, 刷题时的过程模板基本上是:

1、查看题目 2、思考 3、写代码 4、测试 5、提交 6、查看结果

看一道题目,具体是谁看的是不确定的,同样每个人的思考和写出来的代码也是不确定的

测试结果和提交结果也是不确定的。因此我们可以把这1,2,3,4,5,6个过程定义为一个模板方法中的抽象方法部分, 过程的实现交给具体子类去实现

对于一道题目来说, 它是具体的, 因此可以将这道题目定义为抽象模板中的具体方法

模板结构可定义为:



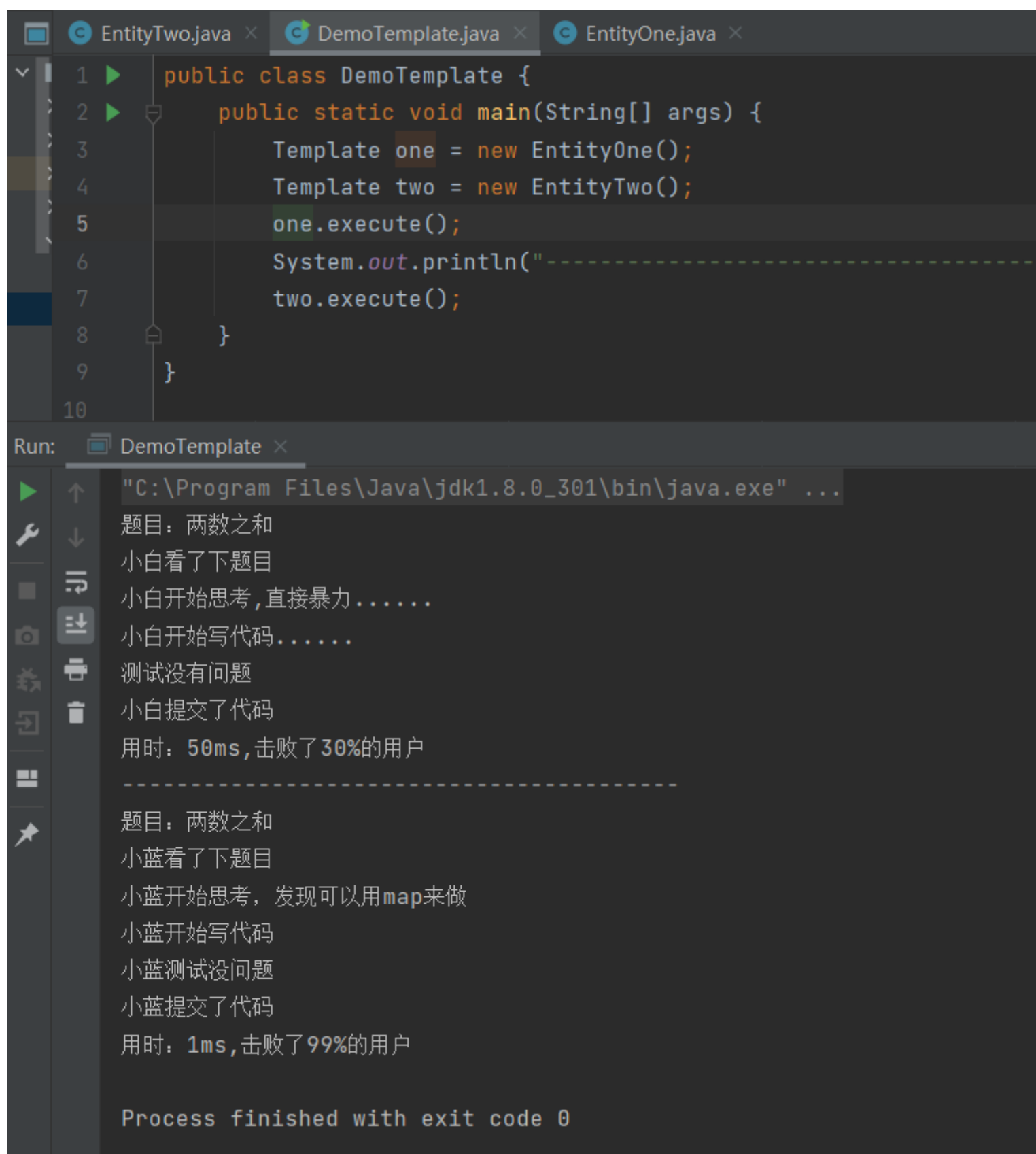
```
1 public abstract class Template {
2     //模拟刷题过程的模板方法
3     public final void execute() {
4         title();
5         look();
6         think();
7         write();
8         test();
9         commit();
10        result();
11    }
12
13    //已经确定的具体的方法(题目)
14    protected void title() {
15        System.out.println("题目: 两数之和");
16    }
17
18    //未确定的抽象的方法
19    //看题目
20    protected abstract void look();
21
22    //思考
23    protected abstract void think();
24
25    //写代码
26    protected abstract void write();
27
28    //测试
29    protected abstract void test();
30
31    //提交
32    protected abstract void commit();
33
34    //查看结果
35    protected abstract void result();
36 }
37
```

具体子类

```
1
2 public class EntityOne extends Template {
3
4     @Override
5     protected void look() {
6         System.out.println("小白看了下题目");
7     }
8
9     @Override
10    protected void think() {
11        System.out.println("小白开始思考,直接暴力.....");
12    }
13
14    @Override
15    protected void write() {
16        System.out.println("小白开始写代码.....");
17    }
18
19    @Override
20    protected void test() {
21        System.out.println("测试没有问题");
22    }
23
24    @Override
25    protected void commit() {
26        System.out.println("提交");
27    }
28
29    @Override
30    protected void result() {
31        System.out.println("用时: 50ms,击败了30%的用户");
32    }
33 }
```

```
1
2 public class EntityTwo extends Template {
3     @Override
4     protected void look() {
5         System.out.println("小蓝看了下题目");
6     }
7
8     @Override
9     protected void think() {
10        System.out.println("小蓝开始思考, 发现可以用map来做");
11    }
12
13    @Override
14    protected void write() {
15        System.out.println("小蓝开始写代码");
16    }
17
18    @Override
19    protected void test() {
20        System.out.println("小蓝测试没问题");
21    }
22
23    @Override
24    protected void commit() {
25        System.out.println("小蓝提交了代码");
26    }
27
28    @Override
29    protected void result() {
30        System.out.println("用时: 1ms, 击败了99%的用户");
31    }
32 }
33
```

运行结果:



The screenshot shows an IDE with three tabs: EntityTwo.java, DemoTemplate.java (active), and EntityOne.java. The DemoTemplate.java file contains the following code:

```
1 public class DemoTemplate {  
2     public static void main(String[] args) {  
3         Template one = new EntityOne();  
4         Template two = new EntityTwo();  
5         one.execute();  
6         System.out.println("-----");  
7         two.execute();  
8     }  
9 }  
10
```

Below the code editor is a Run console window titled "DemoTemplate". It displays the execution output of the program:

```
"C:\Program Files\Java\jdk1.8.0_301\bin\java.exe" ...  
题目：两数之和  
小白看了下题目  
小白开始思考,直接暴力.....  
小白开始写代码.....  
测试没有问题  
小白提交了代码  
用时：50ms,击败了30%的用户  
-----  
题目：两数之和  
小蓝看了下题目  
小蓝开始思考，发现可以用map来做  
小蓝开始写代码  
小蓝测试没问题  
小蓝提交了代码  
用时：1ms,击败了99%的用户  
  
Process finished with exit code 0
```

可以看出，除了抽象类中的具体方法，其他方法都是由子类去实现的，整体算法的流程严格按照，模板方法中的顺序来执行的。

仔细观察运行结果不难看出，如果测试的时候出现了问题，那么应该去修bug，而不是提交。

对于这种情况，我们可以在子类中扩展一个修bug的方法，

代码实现

```
1
2 public class EntityOne extends Template {
3     @Override
4     protected void look() {
5         System.out.println("小白看了下题目");
6     }
7
8     @Override
9     protected void think() {
10        System.out.println("小白开始思考,直接暴力.....");
11    }
12
13    @Override
14    protected void write() {
15        System.out.println("小白开始写代码.....");
16    }
17
18    @Override
19    protected void test() {
20        System.out.println("测试出现了错误!");
21        debug();
22        testTwo();
23    }
24
25    public void debug() {
26        System.out.println("小白尝试修bug.....");
27        System.out.println("小白修好了bug");
28    }
29
30    public void testTwo() {
31        System.out.println("小白再次测试,测试没错误");
32    }
33
34    @Override
35    protected void commit() {
36        System.out.println("小白提交了代码");
37    }
38
39    @Override
40    protected void result() {
41        System.out.println("用时: 50ms,击败了30%的用户");
42    }
43 }
```

运行结果

```
"C:\Program Files\Java\jdk1.8.0_301\bin\java.  
题目：两数之和  
小白看了下题目  
小白开始思考,直接暴力.....  
小白开始写代码.....  
测试出现了错误!  
小白尝试修bug.....  
小白修好了bug  
小白再次测试,测试没错误  
小白提交了代码  
用时：50ms,击败了30%的用户  
-----  
  
Process finished with exit code 0
```

从这里，不难看出模板方法的优点:符合开闭原则，可以通过子类来实现对某些功能的扩展，并且不会影响到原来模板方法

再次观察运行结果，发现可以直接跳过测试，提交代码，但我们之前写的模板方法中并没有跳过测试步骤的这个操作。

所以需要在抽象类添加一个钩子方法来选择是否开启测试这个步骤。

代码实现

四、抽象类添加钩子方法

```
1 public abstract class Template {
2     //模拟刷题过程的模板方法
3     public final void execute() {
4         title();
5         look();
6         think();
7         write();
8         //如果isTest()为true开启test方法
9         if (isTest()) test();
10        commit();
11        result();
12    }
13
14    //已经确定的具体的方法(题目)
15    protected void title() {
16        System.out.println("题目: 两数之和");
17    }
18
19    //未确定的抽象的方法
20    //看题目
21    protected abstract void look();
22
23    //思考
24    protected abstract void think();
25
26    //写代码
27    protected abstract void write();
28
29    //测试
30    protected abstract void test();
31
32    //提交
33    protected abstract void commit();
34
35    //查看结果
36    protected abstract void result();
37
38    //钩子方法, 默认开启
39    protected boolean isTest() {
40        return true;
41    }
42 }
```

子类中实现钩子方法


```
1
2 public class EntityOne extends Template {
3     @Override
4     protected void look() {
5         System.out.println("小白看了下题目");
6     }
7
8     @Override
9     protected void think() {
10        System.out.println("小白开始思考,直接暴力.....");
11    }
12
13
14    @Override
15    protected void write() {
16        System.out.println("小白开始写代码.....");
17    }
18
19    @Override
20    protected void test() {
21        System.out.println("测试出现了错误!");
22        debug();
23        testTwo();
24    }
25
26    public void debug() {
27        System.out.println("小白尝试修bug.....");
28        System.out.println("小白修好了bug");
29    }
30
31    public void testTwo() {
32        System.out.println("小白再次测试,测试没错误");
33    }
34
35    @Override
36    protected void commit() {
37        System.out.println("小白提交了代码");
38    }
39
40    @Override
41    protected void result() {
42        System.out.println("用时: 50ms,击败了30%的用户");
43    }
44
45    @Override
46    protected boolean isTest() {
47        return true;
48    }
49 }
```

```
1 public class EntityTwo extends Template {
2     @Override
3     protected void look() {
4         System.out.println("小蓝看了下题目");
5     }
6
7     @Override
8     protected void think() {
9         System.out.println("小蓝开始思考, 发现可以用map来做");
10    }
11
12    @Override
13    protected void write() {
14        System.out.println("小蓝开始写代码");
15    }
16
17    @Override
18    protected void test() {
19        System.out.println("小蓝测试没问题");
20    }
21
22    @Override
23    protected void commit() {
24        System.out.println("小蓝提交了代码");
25    }
26
27    @Override
28    protected void result() {
29        System.out.println("用时: 1ms, 击败了99%的用户");
30    }
31
32    @Override
33    protected boolean isTest() {
34        return false;
35    }
36 }
```

运行结果

```
题目：两数之和
小白看了下题目
小白开始思考,直接暴力.....
小白开始写代码.....
测试出现了错误!
小白尝试修bug.....
小白修好了bug
小白再次测试,测试没错误
小白提交了代码
用时：50ms,击败了30%的用户

-----

题目：两数之和
小蓝看了下题目
小蓝开始思考，发现可以用map来做
小蓝开始写代码
小蓝提交了代码
用时：1ms,击败了99%的用户

Process finished with exit code 0
```

可以发现，使用钩子方法可以控制，模板方法中的一些执行步骤，合理使用钩子方法可以提高模板方法的灵活性

五、总结

创建流程：

- 1. 创建抽象模板类
- 2. 确定算法整体框架
- 3. 定义具体的方法(不需要改变的方法)
- 4. 定义钩子方法(用来约束方法)
- 5. 声明抽象的方法（需要改变的方法）
- 6. 定义模板方法(方法执行顺序)

应用场景：

- 1. 算法的整体步骤很固定，但其中个别部分容易发生改变时，可以使用模板方法模式，将容易变的部分抽象出来，供子类实现。
- 2. 当多个子类存在相同的方法时，可以将其提取出来并封装到一个公共父类中避免代码重复。
首先，要识别现有代码中的不同之处，并且将不同之处分离为新的操作。
最后，用一个调用这些新的操作的模板方法来替换这些不同的代码。
- 3. 当需要控制子类的扩展时，模板方法只在特定点调用钩子操作，这样就只允许在这些点进行扩展。

优缺点：

优点：

- 提高代码的复用性：将相同部分的代码，封装到了抽象的父类中；
- 提高了代码的扩展性：通过对子类的扩展，增加新的方法；
- 符合开闭原则：通过父类调用子类的操作，通过对子类的扩展来增加新的方法

缺点：

- 增加复杂性：类的数量增加，增加了系统复杂性，引入了抽象类，对于每个实现，都需要定义一个子类；
- 继承缺点：模板方法 主要 通过 继承实现，如果父类增加新的抽象方法，所有的子类都要修改一遍；

