

11 模板方法模式

在JS开发中用到继承场景其实不多，很多时候都喜欢用mix-in的方式来扩展。此外虽然也没有类和继承机制（ES6有了），但是可以通过原型变相继承

本章并非讨论继承，而是基于继承的设计模式--模板方法模式

定义和组成

两部分构成，抽象父类与实现子类。

- 在抽象父类封装子类的算法框架，包括实现公共方法以及封装子类方法的执行顺序
- 子类继承抽象父类，获取算法框架并可以选择性重写

其实模板方法模式是为了解决平行子类存在相同行为与不同行为时，相同部分复用上移的问题，体现了泛化的思想

GPT:

模板方法模式是一种行为设计模式，它定义了一个算法的框架，将算法中的一些步骤延迟到子类中实现。模板方法模式允许子类在不改变算法结构的情况下，重新定义算法的某些步骤。

以下是模板方法模式的关键要点：

1. 模板方法：模板方法是定义算法框架的方法，它包含了一个或多个抽象步骤和具体步骤。抽象步骤由抽象方法或受保护的虚拟方法组成，具体步骤由具体实现提供。模板方法定义了算法的结构和顺序，而将某些步骤的实现委托给子类。
2. 抽象类：抽象类是包含模板方法和抽象步骤的类。抽象类可以定义一些具体的实现，同时也提供抽象方法供子类实现。
3. 具体类：具体类是抽象类的子类，它实现了抽象步骤中的具体方法。

模板方法模式的工作流程如下：

1. 定义算法的框架，并在抽象类中实现模板方法。
2. 将算法中的某些步骤定义为抽象方法或受保护的虚拟方法。
3. 创建具体类，继承抽象类，并实现抽象方法或受保护的虚拟方法。
4. 在具体类中根据需要重写抽象方法，以实现特定的行为。

通过使用模板方法模式，您可以在不改变算法整体结构的情况下，通过子类的具体实现来改变算法中的某些步骤，以满足不同的需求。

模板方法模式在实际应用中广泛使用，例如在框架开发、算法设计、业务流程等方面。它提供了一种灵活而可扩展的方式来定义和管理复杂的算法或流程。

第一个例子-Coffee or Tea

这个例子的原型来自《Head First 设计模式》。这一节我们使用JS来实现。当然过程稍微简化了点

泡一杯咖啡

```
1 class Coffee {
2     constructor(){};
3
4     boilWater = () => {console.log('煮沸水')};
5
6     brewCoffee = () => {console.log('冲泡咖啡')}
7
8     addMilk = () => {console.log('加牛奶')}
9
10    init = () => {
11        this.boilWater();
12        this.brewCoffee();
13        this.addMilk();
14    }
15 }
16
17 const coffee = new Coffee();
18 coffee.init();
```

泡一壶茶

```
1 class Tea {
2     constructor(){};
3
4     boilWater = () => {console.log('煮沸水')};
5
6     steepTea = () => {console.log('冲泡茶')}
7
8     addLemon = () => {console.log('加柠檬')}
9
10    init = () => {
11        this.boilWater();
```

```
12         this.steepTea();
13         this.addLemon();
14     }
15 }
16
17 const tea = new Tea();
18 tea.init();
```

抽象并分离共同点

我们发现泡茶和泡咖啡的不同点在于：原料，冲泡方式，调料不同，但是整个冲泡过程可以整理为三步走：

- 煮沸水
- 冲泡饮料
- 加调料

这里我们通过新的抽象方法名称与抽象父类来概述整个过程：

```
1 class Beverage {
2     constructor(){};
3
4     boilWater = () => {console.log('煮沸水')};
5
6     brew = () => {}
7
8     addCondiments = () => {}
9
10    init = () => {
11        this.boilWater();
12        this.brew();
13        this.addCondiments();
14    }
15 }
16
17 class newCoffee extends Beverage {
18     constructor(){};
19
20     brew = () => {console.log('冲泡咖啡')}
21
22     addCondiments = () => {console.log('加牛奶')}
23 }
24
25 const coffee2 = new newCoffee;
26 coffee2.init();
```

27

28 // newTea同理

上述ES6的继承背后通过原型链的方式来实现继承，最终newCoffee通过父类Beverage复用的共用行为，并通过重写实现了差异化。



上面谁才是模板方法呢？答案是Beverage.prototype.init方法，因为该方法封装了整个子类的算法框架，作为一个算法的模板指导子类方法执行顺序

抽象类

模板方法模式是一种严重依赖抽象类的设计模式，而JS在ES6之前其实没有提供对抽象类的支持，所以这一章重点讨论Java中抽象类的作用以及JS没有抽象类时候做出的让步

抽象类的作用

由于抽象类不能被实例化，通常都是用于被某些具体类继承的

抽象类和接口一样可以用于向上转型，通过把真正的类型隐藏在抽象类或者接口之后，这些对象才可以被互相替换和使用，也让我们的Java程序尽量遵守依赖倒置原则

此外抽象类也表示一种契约，继承了这个抽象类的所有子类都将拥有跟抽象类一致的接口方法，并且不能删除，这在例如模板方法模式等场景下非常有用，可以进行一种模式约束

抽象方法和具体方法

抽象方法被声明在抽象类中，他没有具体的实现，是一个哑巴方法。子类继承时必须重写

而抽象类也可以放置具体实现方法，来方便子类复用，有变动修改抽象类的方法即可，但前提时其具有通用性

用Java实现Coffee和Tea的案例

```
1 abstract class Beverage {
2     // 模板方法，定义了烹饪流程
3     public final void prepareBeverage() {
4         boilWater();
5         brew();
6         pourInCup();
7         addCondiments();
8     }
9
10    // 抽象方法，由子类实现具体的煮沸水的操作
```

```
11     public abstract void boilWater();
12
13     // 抽象方法，由子类实现具体的冲泡操作
14     public abstract void brew();
15
16     // 具体方法，通用的倒入杯子操作
17     public void pourInCup() {
18         System.out.println("Pouring beverage into cup");
19     }
20
21     // 抽象方法，由子类实现具体的加入调料的操作
22     public abstract void addCondiments();
23 }
24
25 class Coffee extends Beverage {
26     @Override
27     public void boilWater() {
28         System.out.println("Boiling water for coffee");
29     }
30
31     @Override
32     public void brew() {
33         System.out.println("Brewing coffee");
34     }
35
36     @Override
37     public void addCondiments() {
38         System.out.println("Adding milk and sugar to coffee");
39     }
40 }
41
42 class Tea extends Beverage {
43     @Override
44     public void boilWater() {
45         System.out.println("Boiling water for tea");
46     }
47
48     @Override
49     public void brew() {
50         System.out.println("Steeping tea leaves");
51     }
52
53     @Override
54     public void addCondiments() {
55         System.out.println("Adding lemon to tea");
56     }
57 }
```

```

58
59 public class Main {
60     public static void main(String[] args) {
61         Beverage coffee = new Coffee();
62         coffee.prepareBeverage();
63
64         System.out.println("-----");
65
66         Beverage tea = new Tea();
67         tea.prepareBeverage();
68     }
69 }
70

```

JS没有抽象类的解决方案

在 JavaScript 中，并没有内置的抽象类和模板方法的概念，但我们可以使用一些技巧来实现类似的效果。

要创建一个抽象类，可以通过以下步骤实现：

1. 创建一个普通的类作为抽象类的基类。
2. 在基类中定义抽象方法，这些方法只有方法签名而没有具体的实现。
3. 在基类中抛出一个错误，提示子类必须实现抽象方法。
4. 子类继承基类，并实现抽象方法。

下面是一个示例，展示如何使用 JavaScript 实现抽象类和模板方法模式：

```

1 class Beverage {
2     constructor() {
3         if (new.target === Beverage) {
4             throw new TypeError('Cannot instantiate abstract class');
5         }
6     }
7
8     prepareBeverage() {
9         this.boilWater();
10        this.brew();
11        this.pourInCup();
12        this.addCondiments();
13    }
14
15    boilWater() {
16        throw new Error('Abstract method not implemented');
17    }

```

```
18
19  brew() {
20      throw new Error('Abstract method not implemented');
21  }
22
23  pourInCup() {
24      console.log('Pouring beverage into cup');
25  }
26
27  addCondiments() {
28      throw new Error('Abstract method not implemented');
29  }
30 }
31
32 class Coffee extends Beverage {
33     boilWater() {
34         console.log('Boiling water for coffee');
35     }
36
37     brew() {
38         console.log('Brewing coffee');
39     }
40
41     addCondiments() {
42         console.log('Adding milk and sugar to coffee');
43     }
44 }
45
46 class Tea extends Beverage {
47     boilWater() {
48         console.log('Boiling water for tea');
49     }
50
51     brew() {
52         console.log('Steeping tea leaves');
53     }
54
55     addCondiments() {
56         console.log('Adding lemon to tea');
57     }
58 }
59
60 const coffee = new Coffee();
61 coffee.prepareBeverage();
62
63 console.log('-----');
64
```

```
65 const tea = new Tea();
66 tea.prepareBeverage();
67
```

在上面的示例中，我们创建了一个基类 `Beverage`，它包含一个抽象方法 `boilWater`、`brew` 和 `addCondiments`。在基类的构造函数中，我们使用 `new.target` 来检查基类是否被直接实例化，如果是则抛出错误。这确保了 `Beverage` 不能直接实例化，只能作为抽象类使用。

`Coffee` 和 `Tea` 是 `Beverage` 的子类，它们继承了抽象类并实现了抽象方法。通过调用 `prepareBeverage` 方法，我们可以触发烹饪流程，具体的步骤由子类实现。

虽然 JavaScript 并没有严格的抽象类和模板方法的语法支持，但使用上述的方式，我们可以实现类似的效果。

希望这个示例对您有帮助。如果您有其他问题，请随时提问。

模板方法模式的使用场景

从大的方面来说，该模式常用于架构师搭建项目或者流程的基础框架，而程序员继承框架的结构之后负责往里面填空，例如Java程序员大多使用过HttpServlet技术来开发项目。

一个HttpServlet包含7个生命周期：

- `doGet`
- `doHead`
- `doPost`
- `doPut`
- `doDelete`
- `doOption`
- `doTrace`

然后该类还提供了一个`service`方法，它就是这里的模板方法，`service`规定了这些`do`方法的执行顺序，而这些`do`方法的具体实现则需要HttpServlet的子类实现

Web开发中也有一些场景，例如在构建系列UI组件的构建过程中：

1. 初始化一个div容器
2. 通过ajax请求拉取相应的数据
3. 把数据渲染到div容器里面，完成组件的构造；
4. 通知用户组件渲染完毕

这样我们可以把上述4个步骤都抽象到父类的模板方法里面，父类中还可以提供第一步和第四步的通用实现

在Web开发中，模板方法模式有一些常见的应用场景，包括：

1. 框架开发：模板方法模式常用于开发框架，其中框架定义了整个应用程序的基本结构和流程，而具体的实现由开发者提供。例如，Web框架可以定义请求处理的流程，包括中间件的执行、路由匹配、控制器的调用等，而具体的处理逻辑由开发者在具体的控制器中实现。
2. 数据库访问：在访问数据库时，可以使用模板方法模式来定义通用的数据库访问流程。例如，可以在抽象类中定义打开数据库连接、执行SQL查询、关闭数据库连接等步骤，具体的数据库操作由不同的子类实现。
3. 用户界面设计：在用户界面的设计中，模板方法模式可以用于创建通用的用户界面元素。例如，可以在抽象类中定义用户界面的创建和渲染流程，包括布局、样式设置、事件处理等，具体的界面元素由不同的子类提供。
4. 流程控制：模板方法模式可以用于控制复杂的业务流程。例如，可以在抽象类中定义业务流程的步骤和顺序，具体的业务逻辑由不同的子类实现。这样可以保持流程的一致性，并允许在子类中定制特定的步骤。

总之，模板方法模式在Web开发中可以帮助提高代码的复用性和可维护性，将通用的流程和结构抽象出来，在子类中提供具体的实现。它适用于那些有固定流程但某些步骤需要定制化的场景。

钩子方法

上述模式的应用其实是要求算法框架在正常状态下是适用于大多数子类的，例如饮料的冲泡过程，但是如果有一些个性的子类不一样，能支持吗？

这里这些“不一样”与”个性“属于一种变化，需要隔离，通常会使用钩子方法”hook“来隔离变化，在父类中容易变化的地方放置钩子，钩子可以有一个默认的实现，而要不要挂钩子由子类决定。

钩子方法的返回结果决定了模板方法后面的执行步骤，这样程序就拥有了变化的可能

```
1  /**
2   * 带钩子方法
3   */
4  class Beverage {
5      constructor(){};
6
7      boilWater = () => {console.log('煮沸水')};
8
9      brew = () => {}
10
11      addCondiments = () => {}
12
```

```

13   customerWantsCondiments = () => {
14       // 钩子方法 - 默认true
15       return true
16   }
17
18   init = () => {
19       this.boilWater();
20       this.brew();
21       if(this.customerWantsCondiments()){
22           this.addCondiments();
23       }
24   }
25 }
26
27 class CoffeeWithHook extends Beverage {
28     boilWater() {
29         console.log('Boiling water for coffee');
30     }
31
32     brew() {
33         console.log('Brewing coffee');
34     }
35
36     addCondiments() {
37         console.log('Adding milk and sugar to coffee');
38     }
39
40     customerWantsCondiments (){
41         return window.confirm('请问需要调料吗? ');
42     }
43 }
44
45 const coffeeWithHook = new CoffeeWithHook();
46 coffeeWithHook.init();

```

好莱坞原则

学习完模板方法模式之后，我们要引入一个新的设计原则 -- 著名的”好莱坞原则“

好莱坞其实也有很多找不到工作的新人，他们经常会主动打电话给演艺公司询问，演艺公司常常回答：不要来找我，有需要我会找你们的

在设计中，这样的规则成为好莱坞原则，我们允许底层组件将自己挂钩到高层组件中，但是高层组件何时以及如何使用就跟演艺公司对待新人一样，”别调用我们，我们需要时候会调用你“

模板方法模式是好莱坞原则的一个经典使用场景，当我们用模板方法模式编写一个程序时候，就意味着子类放弃了对自己的控制权，而是改为父类通知子类，哪些方法应该在什么时候被调用。作为子类，只负责提供一些设计和具体实现上的细节。

除此之外的来包括：

- 发布-订阅模式：发布者会把消息推送给订阅者，取代了原先不断去fetch的消息的形式，就像到了站会有司机叫你一样，不需要你一直问司机到没到站。
- 回调函数：也是不需要去轮询判断要不要调用回调函数，当数据或者指定操作触发时，callback才会执行

总结

模板方法模式是一种典型的通过封装变化提高系统扩展性的设计模式。通过该模式，我们可以把一套逻辑或者框架中的方法种类和执行顺序这部分通用逻辑封装到父类的模板方法里面，而将具体实现的逻辑封装到子类中，通过增加新的子类来给系统实现新的功能

但在JS中，其实不用照Java实现去实现模板方法模式，高阶函数也是一种选择