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，抽象父类与实现字类。

- 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字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

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

```
class Coffee {
    constructor(){};

    boilWater = () => {console.log('煮沸水')};

    brewCoffee = () => {console.log('冲泡咖啡')}

    addMilk = () => {console.log('加牛奶')}

    init = () => {
        this.boilWater();
        this.brewCoffee();
        this.addMilk();
    }
}

const coffee = new Coffee();
```

```
coffee.init();
```

泡一壶茶

```
class Tea {
    constructor(){};

    boilWater = () => {console.log('煮沸水')};

    steepTea = () => {console.log('冲泡茶')}

    addLemon = () => {console.log('加柠檬')}

    init = () => {
        this.boilWater();
        this.steepTea();
        this.addLemon();
    }
}

const tea = new Tea();
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：

```
class Beverage {
    constructor(){};

    boilWater = () => {console.log('煮沸水')};

    brew = () => {}

    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，所以这一张重点讨论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字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

```java
abstract class Beverage {
    // 模板方法，定义了烹饪流程
    public final void prepareBeverage() {
        boilWater();
        brew();
        pourInCup();
        addCondiments();
    }

    // 抽象方法，由子类实现具体的煮沸水的操作
    public abstract void boilWater();

    // 抽象方法，由子类实现具体的冲泡操作
    public abstract void brew();

    // 具体方法，通用的倒入杯子操作
    public void pourInCup() {
        System.out.println("Pouring beverage into cup");
    }

    // 抽象方法，由子类实现具体的加入调料的操作
    public abstract void addCondiments();
}

class Coffee extends Beverage {
    @Override
    public void boilWater() {
        System.out.println("Boiling water for coffee");
    }

    @Override
    public void brew() {
        System.out.println("Brewing coffee");
    }

    @Override
    public void addCondiments() {
        System.out.println("Adding milk and sugar to coffee");
    }
```

```
}

class Tea extends Beverage {
    @Override
    public void boilWater() {
        System.out.println("Boiling water for tea");
    }

    @Override
    public void brew() {
        System.out.println("Steeping tea leaves");
    }

    @Override
    public void addCondiments() {
        System.out.println("Adding lemon to tea");
    }
}

public class Main {
    public static void main(String[] args) {
        Beverage coffee = new Coffee();
        coffee.prepareBeverage();

        System.out.println("----------------------");

        Beverage tea = new Tea();
        tea.prepareBeverage();
    }
}

```

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：

```
class Beverage {
  constructor() {
    if (new.target === Beverage) {
      throw new TypeError('Cannot instantiate abstract class');
    }
  }

  prepareBeverage() {
    this.boilWater();
    this.brew();
    this.pourInCup();
    this.addCondiments();
  }

  boilWater() {
    throw new Error('Abstract method not implemented');
  }

  brew() {
    throw new Error('Abstract method not implemented');
  }

  pourInCup() {
    console.log('Pouring beverage into cup');
  }

  addCondiments() {
    throw new Error('Abstract method not implemented');
  }
}

class Coffee extends Beverage {
  boilWater() {
    console.log('Boiling water for coffee');
  }

  brew() {
    console.log('Brewing coffee');
  }

  addCondiments() {
    console.log('Adding milk and sugar to coffee');
  }
}

class Tea extends Beverage {
  boilWater() {
```

```
    console.log('Boiling water for tea');
  }

  brew() {
    console.log('Steeping tea leaves');
  }

  addCondiments() {
    console.log('Adding lemon to tea');
  }
}

const coffee = new Coffee();
coffee.prepareBeverage();

console.log('----------------------');

const tea = new Tea();
tea.prepareBeverage();

```

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HttpServelet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