第 11 章　模板方法模式

在JavaScript开发中用到继承的场景其实并不是很多，很多时候我们都喜欢用mix-in的方式给对象扩展属性。但这不代表继承在JavaScript里没有用武之地，虽然没有真正的类和继承机制，但我们可以通过原型`prototype`来变相地实现继承。

不过本章并非要讨论继承，而是讨论一种基于继承的设计模式——模板方法（Template Method）模式。

11.1　模板方法模式的定义和组成

模板方法模式是一种只需使用继承就可以实现的非常简单的模式。

模板方法模式由两部分结构组成，第一部分是抽象父类，第二部分是具体的实现子类。通常在抽象父类中封装了子类的算法框架，包括实现一些公共方法以及封装子类中所有方法的执行顺序。子类通过继承这个抽象类，也继承了整个算法结构，并且可以选择重写父类的方法。

假如我们有一些平行的子类，各个子类之间有一些相同的行为，也有一些不同的行为。如果相同和不同的行为都混合在各个子类的实现中，说明这些相同的行为会在各个子类中重复出现。但实际上，相同的行为可以被搬移到另外一个单一的地方，模板方法模式就是为解决这个问题而生的。在模板方法模式中，子类实现中的相同部分被上移到父类中，而将不同的部分留待子类来实现。这也很好地体现了泛化的思想。

11.2 第一个例子——Coffee or Tea

咖啡与茶是一个经典的例子，经常用来讲解模板方法模式，这个例子的原型来自《Head First设计模式》。这一节我们就用JavaScript来实现这个例子。

11.2.1 先泡一杯咖啡

首先，我们先来泡一杯咖啡，如果没有什么太个性化的需求，泡咖啡的步骤通常如下：

(1) 把水煮沸

(2) 用沸水冲泡咖啡

(3) 把咖啡倒进杯子

(4) 加糖和牛奶

通过下面这段代码，我们就能得到一杯香浓的咖啡：

```
var Coffee = function(){};

Coffee.prototype.boilWater = function(){
    console.log( '把水煮沸' );
};

Coffee.prototype.brewCoffeeGriends = function(){
    console.log( '用沸水冲泡咖啡' );
};

Coffee.prototype.pourInCup = function(){
    console.log( '把咖啡倒进杯子' );
};

Coffee.prototype.addSugarAndMilk = function(){
    console.log( '加糖和牛奶' );
};

Coffee.prototype.init = function(){
    this.boilWater();
    this.brewCoffeeGriends();
```

```
    this.pourInCup();
    this.addSugarAndMilk();
};

var coffee = new Coffee();
coffee.init();
```

11.2.2 泡一壶茶

接下来，开始准备我们的茶，泡茶的步骤跟泡咖啡的步骤相差并不大：

(1) 把水煮沸

(2) 用沸水浸泡茶叶

(3) 把茶水倒进杯子

(4) 加柠檬

同样用一段代码来实现泡茶的步骤：

```
var Tea = function(){};

Tea.prototype.boilWater = function(){
    console.log( '把水煮沸' );
};

Tea.prototype.steepTeaBag = function(){
    console.log( '用沸水浸泡茶叶' );
};

Tea.prototype.pourInCup = function(){
    console.log( '把茶水倒进杯子' );
};

Tea.prototype.addLemon = function(){
    console.log( '加柠檬' );
};

Tea.prototype.init = function(){
    this.boilWater();
```

```
    this.steepTeaBag();
    this.pourInCup();
    this.addLemon();
};

var tea = new Tea();
tea.init();
```

11.2.3　分离出共同点

现在我们分别泡好了一杯咖啡和一壶茶，经过思考和比较，我们发现咖啡和茶的冲泡过程是大同小异的，如表11-1所示。

表11-1　咖啡和茶的冲泡过程

泡咖啡	泡茶
把水煮沸	把水煮沸
用沸水冲泡咖啡	用沸水浸泡茶叶
把咖啡倒进杯子	把茶水倒进杯子
加糖和牛奶	加柠檬

我们找到泡咖啡和泡茶主要有以下不同点。

- 原料不同。一个是咖啡，一个是茶，但我们可以把它们都抽象为“饮料”。
- 泡的方式不同。咖啡是冲泡，而茶叶是浸泡，我们可以把它们都抽象为“泡”。
- 加入的调料不同。一个是糖和牛奶，一个是柠檬，但我们可以把它们

都抽象为“调料”。

经过抽象之后，不管是泡咖啡还是泡茶，我们都能整理为下面四步：

(1) 把水煮沸

(2) 用沸水冲泡饮料

(3) 把饮料倒进杯子

(4) 加调料

所以，不管是冲泡还是浸泡，我们都能给它一个新的方法名称，比如说brew()。同理，不管是加糖和牛奶，还是加柠檬，我们都可以称之为addCondiments()。

让我们忘记最开始创建的Coffee 类和Tea 类。 现在可以创建一个抽象父类来表示泡一杯饮料的整个过程。不论是Coffee ， 还是Tea ， 都被我们用Beverage 来表示，代码如下：

```
var Beverage = function(){};

Beverage.prototype.boilWater = function(){
    console.log( '把水煮沸' );
};

Beverage.prototype.brew = function(){};       // 空方法，应该由子类重写

Beverage.prototype.pourInCup = function(){};     // 空方法，应该由子类重写

Beverage.prototype.addCondiments = function(){};     // 空方法，应该由子类重

Beverage.prototype.init = function(){
    this.boilWater();
    this.brew();
    this.pourInCup();
    this.addCondiments();
};
```

11.2.4 创建Coffee子类和Tea子类

现在创建一个Beverage 类的对象对我们来说没有意义，因为世界上能喝的东西没有一种真正叫“饮料”的，饮料在这里还只是一个抽象的存在。接下来我们要创建咖啡类和茶类，并让它们继承饮料类：

```
var Coffee = function(){};

Coffee.prototype = new Beverage();
```

接下来要重写抽象父类中的一些方法，只有“把水煮沸”这个行为可以直接使用父类Beverage 中的boilWater 方法，其他方法都需要在Coffee 子类中重写，代码如下：

```
Coffee.prototype.brew = function(){
    console.log( '用沸水冲泡咖啡' );
};

Coffee.prototype.pourInCup = function(){
    console.log( '把咖啡倒进杯子' );
};

Coffee.prototype.addCondiments = function(){
    console.log( '加糖和牛奶' );
};

var Coffee = new Coffee();
Coffee.init();
```

至此我们的Coffee 类已经完成了，当调用coffee 对象的init 方法时，由于coffee 对象和Coffee 构造器的原型prototype 上都没有对应的init 方法，所以该请求会顺着原型链，被委托给Coffee 的“父类”Beverage 原型上的init 方法。

而Beverage.prototype.init 方法中已经规定好了泡饮料的顺序，所

以我们能成功地泡出一杯咖啡，代码如下：

```
Beverage.prototype.init = function(){
    this.boilWater();
    this.brew();
    this.pourInCup();
    this.addCondiments();
};
```

接下来照葫芦画瓢，来创建我们的Tea 类：

```
var Tea = function(){};

Tea.prototype = new Beverage();

Tea.prototype.brew = function(){
    console.log( '用沸水浸泡茶叶' );
};

Tea.prototype.pourInCup = function(){
    console.log( '把茶倒进杯子' );
};

Tea.prototype.addCondiments = function(){
    console.log( '加柠檬' );
};

var tea = new Tea();
tea.init();
```

本章一直讨论的是模板方法模式，那么在上面的例子中，到底谁才是所谓的模板方法呢？答案是Beverage.prototype.init 。

Beverage.prototype.init 被称为模板方法的原因是，该方法中封装了子类的算法框架，它作为一个算法的模板，指导子类以何种顺序去执行哪些方法。在Beverage.prototype.init 方法中，算法内的每一个步骤都清楚地展示在我们眼前。

11.3 抽象类

首先要说明的是，模板方法模式是一种严重依赖抽象类的设计模式。JavaScript在语言层面并没有提供对抽象类的支持，我们也很难模拟抽象类的实现。这一节我们将着重讨论Java中抽象类的作用，以及JavaScript没有抽象类时所做出的让步和变通。

11.3.1 抽象类的作用

在Java中，类分为两种，一种为具体类，另一种为抽象类。具体类可以被实例化，抽象类不能被实例化。要了解抽象类不能被实例化的原因，我们可以思考“饮料”这个抽象类。

想象这样一个场景：我们口渴了，去便利店想买一瓶饮料，我们不能直接跟店员说：“来一瓶饮料。”如果我们这样说了，那么店员接下来肯定会问：“要什么饮料？”饮料只是一个抽象名词，只有当我们真正明确了的饮料类型之后，才能得到一杯咖啡、茶、或者可乐。

由于抽象类不能被实例化，如果有人编写了一个抽象类，那么这个抽象类一定是用来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（可参考1.3节关于多态的内容），在静态类型语言中，编译器对类型的检查总是一个绕不过的话题与困扰。虽然类型检查可以提高程序的安全性，但繁琐而严格的类型检查也时常会让程序员觉得麻烦。把对象的真正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使用。这可以让我们的Java程序尽量遵守依赖倒置原则。

除了用于向上转型，抽象类也可以表示一种契约。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，抽象类的主要作用就是为它的子类定义这些公共接口。如果我们在子类中删掉了这些方法中的某一个，那么将不能通过编译器的检查，这在某些场景下是非常有用的，比如我们本章讨论的模板方法模式，`Beverage` 类的`init` 方法里规定了冲泡一杯饮料的顺序如下：

```
this.boilWater();     // 把水煮沸
this.brew();      // 用水泡原料
```

```
this.pourInCup();     // 把原料倒进杯子
this.addCondiments();     // 添加调料
```

如果在Coffee 子类中没有实现对应的brew 方法，那么我们百分之百得不到一杯咖啡。既然父类规定了子类的方法和执行这些方法的顺序，子类就应该拥有这些方法，并且提供正确的实现。

11.3.2 抽象方法和具体方法

抽象方法被声明在抽象类中，抽象方法并没有具体的实现过程，是一些“哑”方法。比如Beverage 类中的brew 方法、pourInCup 方法和addCondiments 方法，都被声明为抽象方法。当子类继承了这个抽象类时，必须重写父类的抽象方法。

除了抽象方法之外，如果每个子类中都有一些同样的具体实现方法，那这些方法也可以选择放在抽象类中，这可以节省代码以达到复用的效果，这些方法叫作具体方法。当代码需要改变时，我们只需要改动抽象类里的具体方法就可以了。比如饮料中的boilWater 方法，假设冲泡所有的饮料之前，都要先把水煮沸，那我们自然可以把boilWater 方法放在抽象类Beverage 中。

11.3.3 用Java实现Coffee or Tea的例子

下面我们尝试着把Coffee和Tea的例子换成Java代码，这有助于我们理解抽象类的意义。

```
// Java代码

public abstract class Beverage {     // 饮料抽象类
    final void init(){     // 模板方法
        boilWater();
        brew();
        pourInCup();
        addCondiments();
    }

    void boilWater(){         // 具体方法boilWater
```

```
        System.out.println( "把水煮沸" );
    }

    abstract void brew();         // 抽象方法brew
    abstract void addCondiments();         // 抽象方法addCondiments
    abstract void pourInCup();         // 抽象方法pourInCup
}


public class Coffee extends Beverage{         // Coffee类
    @Override
    void brew() {    // 子类中重写brew方法
        System.out.println( "用沸水冲泡咖啡" );
    }

    @Override
    void pourInCup(){    // 子类中重写pourInCup方法
        System.out.println( "把咖啡倒进杯子" );
    }

    @Override
    void addCondiments() {         // 子类中重写addCondiments方法
        System.out.println( "加糖和牛奶" );
    }
}


public class Tea extends Beverage{         // Tea类
    @Override
    void brew() {         // 子类中重写brew方法
        System.out.println( "用沸水浸泡茶叶" );
    }

    @Override
    void pourInCup(){         // 子类中重写pourInCup方法
        System.out.println( "把茶倒进杯子" );
    }

    @Override
    void addCondiments() {         // 子类中重写addCondiments方法
        System.out.println( "加柠檬" );
    }
}

public class Test {

    private static void prepareRecipe( Beverage beverage ){
        beverage.init();
    }
```

```
    public static void main( String args[] ){
         Beverage coffee = new Coffee();   // 创建coffee对象
         prepareRecipe( coffee );   // 开始泡咖啡
        // 把水煮沸
        // 用沸水冲泡咖啡
        // 把咖啡倒进杯子
        // 加糖和牛奶

        Beverage tea = new Tea();   // 创建tea对象
        prepareRecipe( tea );   // 开始泡茶
        // 把水煮沸
        // 用沸水浸泡茶叶
        // 把茶倒进杯子
        // 加柠檬
    }
}
```

11.3.4 JavaScript没有抽象类的缺点和解决方案

JavaScript并没有从语法层面提供对抽象类的支持。抽象类的第一个作用是隐藏对象的具体类型，由于JavaScript是一门“类型模糊”的语言，所以隐藏对象的类型在JavaScript中并不重要。

另一方面，当我们在JavaScript中使用原型继承来模拟传统的类式继承时，并没有编译器帮助我们进行任何形式的检查，我们也没有办法保证子类会重写父类中的“抽象方法”。

我们知道，`Beverage.prototype.init` 方法作为模板方法，已经规定了子类的算法框架，代码如下：

```
Beverage.prototype.init = function(){
    this.boilWater();
    this.brew();
    this.pourInCup();
    this.addCondiments();
};
```

如果我们的Coffee 类或者Tea 类忘记实现这4个方法中的一个呢？拿brew 方法举例，如果我们忘记编写Coffee.prototype.brew 方法，那么当请求coffee 对象的brew 时，请求会顺着原型链找到Beverage “父类”对应的Beverage.prototype.brew 方法，
而Beverage.prototype.brew 方法到目前为止是一个空方法，这显然是不能符合我们需要的。

在Java中编译器会保证子类会重写父类中的抽象方法，但在JavaScript中却没有进行这些检查工作。我们在编写代码的时候得不到任何形式的警告，完全寄托于程序员的记忆力和自觉性是很危险的，特别是当我们使用模板方法模式这种完全依赖继承而实现的设计模式时。

下面提供两种变通的解决方案。

- 第1种方案是用鸭子类型来模拟接口检查，以便确保子类中确实重写了父类的方法。但模拟接口检查会带来不必要的复杂性，而且要求程序员主动进行这些接口检查，这就要求我们在业务代码中添加一些跟业务逻辑无关的代码。

- 第2种方案是让Beverage.prototype.brew 等方法直接抛出一个异常，如果因为粗心忘记编写Coffee.prototype.brew 方法，那么至少我们会在程序运行时得到一个错误：

```
Beverage.prototype.brew = function(){
    throw new Error( '子类必须重写brew方法'  );
};


Beverage.prototype.pourInCup = function(){
    throw new Error( '子类必须重写pourInCup方法'  );
};


Beverage.prototype.addCondiments = function(){
    throw new Error( '子类必须重写addCondiments方法'  );
};
```

第2种解决方案的优点是实现简单，付出的额外代价很少；缺点是我们得到错误信息的时间点太靠后。

我们一共有3次机会得到这个错误信息，第1次是在编写代码的时候，通过编译器的检查来得到错误信息；第2次是在创建对象的时候用鸭子类型来进行“接口检查”；而目前我们不得不利用最后一次机会，在程序运行过程中才知道哪里发生了错误。

11.4 模板方法模式的使用场景

从大的方面来讲，模板方法模式常被架构师用于搭建项目的框架，架构师定好了框架的骨架，程序员继承框架的结构之后，负责往里面填空，比如Java程序员大多使用过HttpServlet技术来开发项目。

一个基于HttpServlet的程序包含7个生命周期，这7个生命周期分别对应一个do 方法。

```
doGet()
doHead()
doPost()
doPut()
doDelete()
doOption()
doTrace()
```

HttpServlet 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 的子类来提供。

在Web开发中也能找到很多模板方法模式的适用场景，比如我们在构建一系列的UI组件，这些组件的构建过程一般如下所示：

(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。

我们看到，任何组件的构建都遵循上面的4步，其中第(1)步和第(4)步是相同的。第(2)步不同的地方只是请求ajax的远程地址，第(3)步不同的地方是渲染数据的方式。

于是我们可以把这4个步骤都抽象到父类的模板方法里面，父类中还可以顺便提供第(1)步和第(4)步的具体实现。当子类继承这个父类之后，会重写模板方法里面的第(2)步和第(3)步。

11.5 钩子方法

通过模板方法模式，我们在父类中封装了子类的算法框架。这些算法框架在正常状态下是适用于大多数子类的，但如果有一些特别“个性”的子类呢？比如我们在饮料类Beverage 中封装了饮料的冲泡顺序：

(1) 把水煮沸

(2) 用沸水冲泡饮料

(3) 把饮料倒进杯子

(4) 加调料

这4个冲泡饮料的步骤适用于咖啡和茶，在我们的饮料店里，根据这4个步骤制作出来的咖啡和茶，一直顺利地提供给绝大部分客人享用。但有一些客人喝咖啡是不加调料（糖和牛奶）的。既然Beverage 作为父类，已经规定好了冲泡饮料的4个步骤，那么有什么办法可以让子类不受这个约束呢？

钩子方法 （hook ）可以用来解决这个问题，放置钩子是隔离变化的一种常见手段。我们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，究竟要不要“挂钩”，这由子类自行决定。钩子方法的返回结果决定了模板方法后面部分的执行步骤，也就是程序接下来的走向，这样一来，程序就拥有了变化的可能。

在这个例子里，我们把挂钩的名字定为customerWantsCondiments ，接下来将挂钩放入Beverage 类，看看我们如何得到一杯不需要糖和牛奶的咖啡，代码如下：

```
var Beverage = function(){};

Beverage.prototype.boilWater = function(){
    console.log( '把水煮沸' );
};

Beverage.prototype.brew = function(){
    throw new Error( '子类必须重写brew方法' );
};
```

```
Beverage.prototype.pourInCup = function(){
    throw new Error( '子类必须重写pourInCup方法' );
};

Beverage.prototype.addCondiments = function(){
    throw new Error( '子类必须重写addCondiments方法' );
};

Beverage.prototype.customerWantsCondiments = function(){
    return true;    // 默认需要调料
};

Beverage.prototype.init = function(){
    this.boilWater();
    this.brew();
    this.pourInCup();
    if ( this.customerWantsCondiments() ){    // 如果挂钩返回true，则需要调
        this.addCondiments();
    }
};


var CoffeeWithHook = function(){};

CoffeeWithHook.prototype = new Beverage();

CoffeeWithHook.prototype.brew = function(){
    console.log( '用沸水冲泡咖啡' );
};

CoffeeWithHook.prototype.pourInCup = function(){
    console.log( '把咖啡倒进杯子' );
};

CoffeeWithHook.prototype.addCondiments = function(){
    console.log( '加糖和牛奶' );
};

CoffeeWithHook.prototype.customerWantsCondiments = function(){
    return window.confirm( '请问需要调料吗？' );
};

var coffeeWithHook = new CoffeeWithHook();
coffeeWithHook.init();
```

11.6　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”。

好莱坞无疑是演员的天堂，但好莱坞也有很多找不到工作的新人演员，许多新人演员在好莱坞把简历递给演艺公司之后就只有回家等待电话。有时候该演员等得不耐烦了，给演艺公司打电话询问情况，演艺公司往往这样回答：“不要来找我，我会给你打电话。”

在设计中，这样的规则就称为好莱坞原则。在这一原则的指导下，我们允许底层组件将自己挂钩到高层组件中，而高层组件会决定什么时候、以何种方式去使用这些底层组件，高层组件对待底层组件的方式，跟演艺公司对待新人演员一样，都是“别调用我们，我们会调用你”。

模板方法模式是好莱坞原则的一个典型使用场景，它与好莱坞原则的联系非常明显，当我们用模板方法模式编写一个程序时，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上的细节。

除此之外，好莱坞原则还常常应用于其他模式和场景，例如发布-订阅模式和回调函数。

- 发布—订阅模式

  在发布—订阅模式中，发布者会把消息推送给订阅者，这取代了原先不断去fetch消息的形式。例如假设我们乘坐出租车去一个不了解的地方，除了每过5秒钟就问司机“是否到达目的地”之外，还可以在车上美美地睡上一觉，然后跟司机说好，等目的地到了就叫醒你。这也相当于好莱坞原则中提到的“别调用我们，我们会调用你”。

- 回调函数

  在ajax异步请求中，由于不知道请求返回的具体时间，而通过轮询去判断是否返回数据，这显然是不理智的行为。所以我们通常会把接下来的操作放在回调函数中，传入发起ajax异步请求的函数。当数据返回之后，这个回调函数才被执行，这也是好莱坞原则的一种体现。把

需要执行的操作封装在回调函数里，然后把主动权交给另外一个函数。至于回调函数什么时候被执行，则是另外一个函数控制的。

11.7 真的需要“继承”吗

模板方法模式是基于继承的一种设计模式，父类封装了子类的算法框架和方法的执行顺序，子类继承父类之后，父类通知子类执行这些方法，好莱坞原则很好地诠释了这种设计技巧，即高层组件调用底层组件。

本章我们通过模板方法模式，编写了一个Coffee or Tea的例子。模板方法模式是为数不多的基于继承的设计模式，但JavaScript语言实际上没有提供真正的类式继承，继承是通过对象与对象之间的委托来实现的。也就是说，虽然我们在形式上借鉴了提供类式继承的语言，但本章学习到的模板方法模式并不十分正宗。而且在JavaScript这般灵活的语言中，实现这样一个例子，是否真的需要继承这种重武器呢？

在好莱坞原则的指导之下，下面这段代码可以达到和继承一样的效果。

```
var Beverage = function( param ){

    var boilWater = function(){
        console.log( '把水煮沸' );
    };

    var brew = param.brew || function(){
        throw new Error( '必须传递brew方法' );
    };

    var pourInCup = param.pourInCup || function(){
        throw new Error( '必须传递pourInCup方法' );
    };

    var addCondiments = param.addCondiments || function(){
        throw new Error( '必须传递addCondiments方法' );
    };

    var F = function(){};

    F.prototype.init = function(){
        boilWater();
        brew();
        pourInCup();
        addCondiments();
    };

    return F;
```

```
};

var Coffee = Beverage({
    brew: function(){
        console.log( '用沸水冲泡咖啡' );
    },
    pourInCup: function(){
         console.log( '把咖啡倒进杯子' );
     },
     addCondiments: function(){
         console.log( '加糖和牛奶' );
     }
});

var Tea = Beverage({
    brew: function(){
        console.log( '用沸水浸泡茶叶' );
    },
    pourInCup: function(){
         console.log( '把茶倒进杯子' );
     },
     addCondiments: function(){
         console.log( '加柠檬' );
     }
});


var coffee = new Coffee();
coffee.init();

var tea = new Tea();
tea.init();
```

在这段代码中，我们把brew 、pourInCup 、addCondiments 这些方法依次传入Beverage 函数，Beverage 函数被调用之后返回构造器F。F类中包含了“模板方法”F.prototype.init 。跟继承得到的效果一样，该“模板方法”里依然封装了饮料子类的算法框架。