[S.O.L.I.D五大原则之开闭原则OCP](http://www.cnblogs.com/TomXu/archive/2012/01/09/2306329.html)

**前言**

开闭原则的描述是：

Software entities (classes, modules, functions, etc.) should be open for extension but closed for modification.  
软件实体（类，模块，方法等等）应当对扩展开放，对修改关闭，即软件实体应当在不修改的前提下扩展。

open for extension（对扩展开放）的意思是说当新需求出现的时候，可以通过扩展现有模型达到目的。而Close for modification（对修改关闭）的意思是说不允许对该实体做任何修改，说白了，就是这些需要执行多样行为的实体应该设计成不需要修改就可以实现各种的变化，坚持开闭原则有利于用最少的代码进行项目维护。

英文原文：http://freshbrewedcode.com/derekgreer/2011/12/19/solid-javascript-the-openclosed-principle/

**问题代码**

为了直观地描述，我们来举个例子演示一下，下属代码是动态展示question列表的代码（没有使用开闭原则）。

// 问题类型  
var AnswerType = {  
 Choice: 0,  
 Input: 1  
};  
  
// 问题实体  
function question(label, answerType, choices) {  
 return {  
 label: label,  
 answerType: answerType,  
 choices: choices // 这里的choices是可选参数  
 };  
}  
  
var view = (function () {  
 // render一个问题  
 function renderQuestion(target, question) {  
 var questionWrapper = document.createElement('div');  
 questionWrapper.className = 'question';  
  
 var questionLabel = document.createElement('div');  
 questionLabel.className = 'question-label';  
 var label = document.createTextNode(question.label);  
 questionLabel.appendChild(label);  
  
 var answer = document.createElement('div');  
 answer.className = 'question-input';  
  
 // 根据不同的类型展示不同的代码：分别是下拉菜单和输入框两种  
 if (question.answerType === AnswerType.Choice) {  
 var input = document.createElement('select');  
 var len = question.choices.length;  
 for (var i = 0; i < len; i++) {  
 var option = document.createElement('option');  
 option.text = question.choices[i];  
 option.value = question.choices[i];  
 input.appendChild(option);  
 }  
 }  
 else if (question.answerType === AnswerType.Input) {  
 var input = document.createElement('input');  
 input.type = 'text';  
 }  
  
 answer.appendChild(input);  
 questionWrapper.appendChild(questionLabel);  
 questionWrapper.appendChild(answer);  
 target.appendChild(questionWrapper);  
 }  
  
 return {  
 // 遍历所有的问题列表进行展示  
 render: function (target, questions) {  
 for (var i = 0; i < questions.length; i++) {  
 renderQuestion(target, questions[i]);  
 };  
 }  
 };  
})();  
  
var questions = [  
 question('Have you used tobacco products within the last 30 days?', AnswerType.Choice, ['Yes', 'No']),  
 question('What medications are you currently using?', AnswerType.Input)  
 ];  
  
var questionRegion = document.getElementById('questions');  
view.render(questionRegion, questions);

上面的代码，view对象里包含一个render方法用来展示question列表，展示的时候根据不同的question类型使用不同的展示方式，一个question包含一个label和一个问题类型以及choices的选项（如果是选择类型的话）。如果问题类型是Choice那就根据选项生产一个下拉菜单，如果类型是Input，那就简单地展示input输入框。

该代码有一个限制，就是如果再增加一个question类型的话，那就需要再次修改renderQuestion里的条件语句，这明显违反了开闭原则。

**重构代码**

让我们来重构一下这个代码，以便在出现新question类型的情况下允许扩展view对象的render能力，而不需要修改view对象内部的代码。

先来创建一个通用的questionCreator函数：

function questionCreator(spec, my) {  
 var that = {};  
  
 my = my || {};  
 my.label = spec.label;  
  
 my.renderInput = function () {  
 throw "not implemented";   
 // 这里renderInput没有实现，主要目的是让各自问题类型的实现代码去覆盖整个方法  
 };  
  
 that.render = function (target) {  
 var questionWrapper = document.createElement('div');  
 questionWrapper.className = 'question';  
  
 var questionLabel = document.createElement('div');  
 questionLabel.className = 'question-label';  
 var label = document.createTextNode(spec.label);  
 questionLabel.appendChild(label);  
  
 var answer = my.renderInput();  
 // 该render方法是同样的粗合理代码  
 // 唯一的不同就是上面的一句my.renderInput()  
 // 因为不同的问题类型有不同的实现  
  
 questionWrapper.appendChild(questionLabel);  
 questionWrapper.appendChild(answer);  
 return questionWrapper;  
 };  
  
 return that;  
}

该代码的作用组合要是render一个问题，同时提供一个未实现的renderInput方法以便其他function可以覆盖，以使用不同的问题类型，我们继续看一下每个问题类型的实现代码：

function choiceQuestionCreator(spec) {  
  
 var my = {},  
that = questionCreator(spec, my);  
   
 // choice类型的renderInput实现  
 my.renderInput = function () {  
 var input = document.createElement('select');  
 var len = spec.choices.length;  
 for (var i = 0; i < len; i++) {  
 var option = document.createElement('option');  
 option.text = spec.choices[i];  
 option.value = spec.choices[i];  
 input.appendChild(option);  
 }  
  
 return input;  
 };  
  
 return that;  
}  
  
function inputQuestionCreator(spec) {  
  
 var my = {},  
that = questionCreator(spec, my);  
  
 // input类型的renderInput实现  
 my.renderInput = function () {  
 var input = document.createElement('input');  
 input.type = 'text';  
 return input;  
 };  
  
 return that;  
}

choiceQuestionCreator函数和inputQuestionCreator函数分别对应下拉菜单和input输入框的renderInput实现，通过内部调用统一的questionCreator(spec, my)然后返回that对象（同一类型哦）。

view对象的代码就很固定了。

var view = {  
 render: function(target, questions) {  
 for (var i = 0; i < questions.length; i++) {  
 target.appendChild(questions[i].render());  
 }  
 }  
};

所以我们声明问题的时候只需要这样做，就OK了：

var questions = [  
 choiceQuestionCreator({  
 label: 'Have you used tobacco products within the last 30 days?',  
 choices: ['Yes', 'No']  
　　}),  
 inputQuestionCreator({  
 label: 'What medications are you currently using?'  
　　})  
 ];

最终的使用代码，我们可以这样来用：

var questionRegion = document.getElementById('questions');  
  
view.render(questionRegion, questions);

上面的代码里应用了一些技术点，我们来逐一看一下：

1. 首先，questionCreator方法的创建，可以让我们使用[模板方法模式](http://en.wikipedia.org/wiki/Template_method_pattern" \t "_blank)将处理问题的功能delegat给针对每个问题类型的扩展代码renderInput上。
2. 其次，我们用一个私有的spec属性替换掉了前面question方法的构造函数属性，因为我们封装了render行为进行操作，不再需要把这些属性暴露给外部代码了。
3. 第三，我们为每个问题类型创建一个对象进行各自的代码实现，但每个实现里都必须包含renderInput方法以便覆盖questionCreator方法里的renderInput代码，这就是我们常说的[策略模式](http://en.wikipedia.org/wiki/Strategy_pattern" \t "_blank)。

通过重构，我们可以去除不必要的问题类型的枚举AnswerType，而且可以让choices作为choiceQuestionCreator函数的必选参数（之前的版本是一个可选参数）。

**总结**

重构以后的版本的view对象可以很清晰地进行新的扩展了，为不同的问题类型扩展新的对象，然后声明questions集合的时候再里面指定类型就行了，view对象本身不再修改任何改变，从而达到了开闭原则的要求。