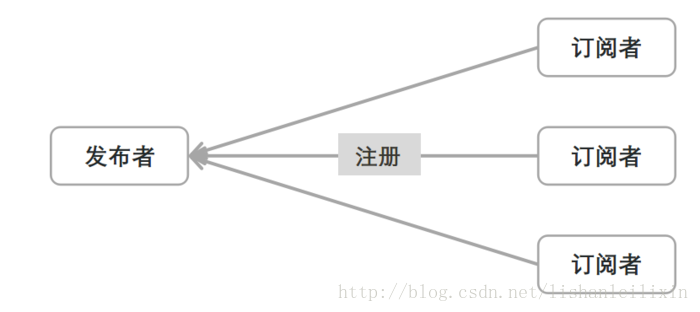
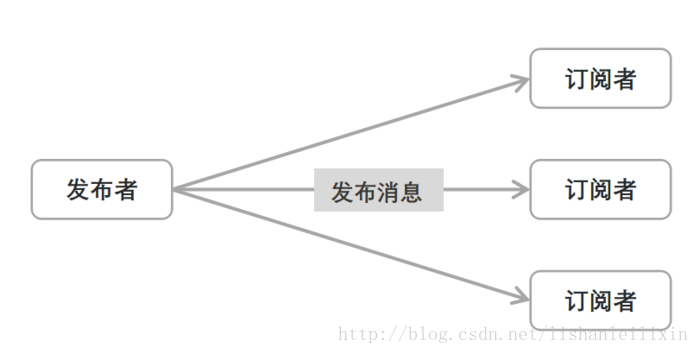
**vue双向绑定原理 及 计算属性**

vue的双向绑定是由**数据劫持**结合**发布者－订阅者模式**实现的，实际就是通过Object.defineProperty()来劫持对象属性的setter和getter操作，在数据变动时做需要做的事情。





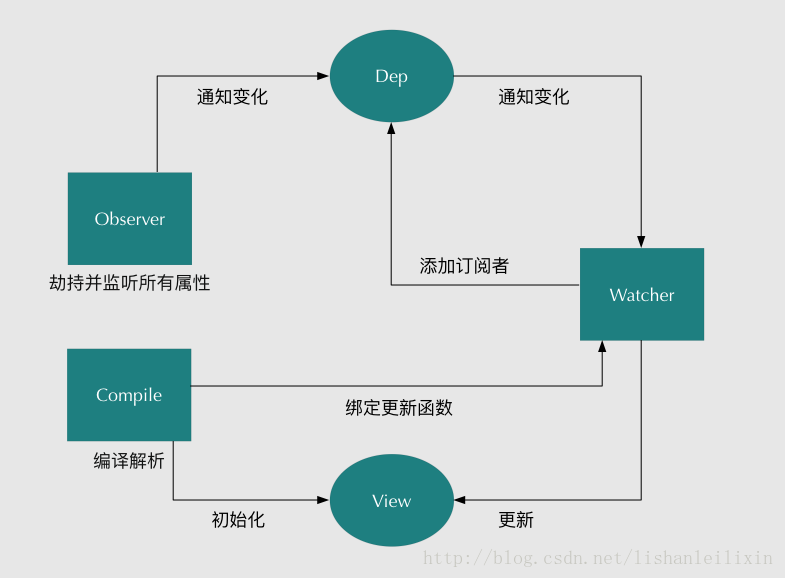
既然是 **数据劫持**、**发布者－订阅者** 模式，那就需要 劫持的函数、发布者和订阅者了:

**\*** 实现一个**监听器Observer**(劫持者)。他的作用是如果被监听的数据有变动就通知发布者。

**\*** 实现一个**发布者 Dep**。他的作用是收到数据变化的通知，并通知所有对应的订阅者

**\*** 实现一个**订阅者 Watcher**。他的作用是收到数据变化的通知并执行对应函数

**\*** 实现一个**解析器** **Compile。**他的作用是解析节点中的指令，初始化模板数据等操作



1.首先先实现对源数据进行监听(劫持)的Observer。数据监听的核心方法是 [Object.defineproperty();](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Reference/Global_Objects/Object/defineProperty) 只要遍历所有的属性，对其进行处理即可。(写在Observer.js中)

*/\*\**

*\**

*\* @param {Object} data 当前遍历到的节点*

*\*/*

function observe(*data*) {

// 如果当前节点不含子节点则返回，进行defineproperty处理

if (!data || typeof data !== 'object') return;

Object.keys(data).forEach((*key*) => defineReactive(data, key, data[key]));

}

*/\*\**

*\**

*\* @param {Object} data 当前遍历到的节点*

*\* @param {string} key 属性名*

*\* @param {\*} value 属性值*

*\*/*

function defineReactive(*data*, *key*, *value*) {

observe(value); *// 递归遍历*

Object.defineProperty(data, key, {

enumerable: true, // 属性可以被循环

configurable: true, // 属性可以再修改

get() {

*// 添加订阅者...*

*// 添加计算属性...*

return value;

},

set(*newValue*) {

*// 如果值未改变*

if (value === newValue) return;

*// 如果值改变*

value = newValue;

console.log(`属性${key}的值发生变化，现在为：${value}`);

}

})

}

// 现在可以进行验证

var library = {

book1: {

name: ''

},

book2: ''

};

observe(library);

library.book1.name = 'vue权威指南'; *// 属性name的值发生变化，现在为：vue权威指南“vue权威指南”*

library.book2 = '没有此书籍'; *// 属性book2的值发生变化，现在为：没有此书籍*

2.通过observe 循环每个属性，再通过defineReactive 进行处理get 和 set便能达到监听数据的需求。此外，我们应该要有一个’人’知道谁订阅了这个数据，所以我们便请出了 发布者Dep。我们修改一下以上代码：

*/\* 这里是发布者 \*/*

function Dep() {

this.subs = []; *// 保存订阅者*

}

Dep.prototype = {

*/\* 添加一个订阅者 \*/*

addSub(*sub*) {

this.subs.push(sub);

},

*/\* 遍历并通知所有订阅者 \*/*

notify() {

this.subs.forEach(*sub* => sub.update());

}

}

Dep.target = null; *// 用来缓存订阅者 Watcher*

*/\*\**

*\**

*\* @param {Object} data 当前遍历到的节点*

*\*/*

function observe(*data*) {

*// 如果当前节点不含子节点则返回，进行defineproperty处理*

if (!data || typeof data !== 'object') return;

Object.keys(data).forEach((*key*) => defineReactive(data, key, data[key]));

}

*/\*\**

*\**

*\* @param {Object} data 当前遍历到的节点*

*\* @param {string} key 属性名*

*\* @param {\*} value 属性值*

*\*/*

function defineReactive(*data*, *key*, *value*) {

observe(value); *// 递归遍历*

var dep = new Dep(); *// 消息订阅器*

Object.defineProperty(data, key, {

enumerable: true, *// 属性可以被循环*

configurable: true, *// 属性可以被修改*

get() {

if (是否添加) { *// 如果是初始化的watcher，则添加到dep数组中*

dep.addSub(watcher); *// 添加watcher*

}

*/\**

*if(是否添加) {*

*// 添加计算属性...*

*}*

*\*/*

return value;

},

set(*newValue*) {

*// 如果值未改变*

if (value === newValue) return;

*// 如果值改变*

value = newValue;

dep.notify(); *// 通知订阅者执行相应更新函数*

*//console.log(`属性${key}的值发生变化，现在为：${value}`);*

}

})

}

在get()中，放置了一个if判断。让一个订阅者(Watcher)被创建时，会以一种方式(下文讲解)触发get()，使订阅者自身添加到Dep()中。在set()里，放置了Dep的notify()方法。当数据改变时触发set()便会一同触发notify()方法，通知所有的订阅者了。

3. 至此，**监听器(Observer)**和**发布者(Dep)**的实现就告一段落了。接下来我们实现**订阅者(Watcher)**。

因为我们在监听器Observer 的get()中放置了添加订阅者的方法，所以我们需要去触发get()，那该怎么触发呢？其实很简单，get()的功能就是在获取属性值的时候执行，所以我们只需要在初始化订阅者的时候获取一下属性值就行。此外get()中的if我们可以通过Dep.target来判断，所以在初始化的时候还要将Dep.target赋值，其实也就是将订阅者自己赋给Dep.target，再添加成功后再释放即可。(写在Watcher.js中)

*/\*\**

*\**

*\* @param {上下文环境} vm 执行selfVue的作用域*

*\* @param {string} key 属性名*

*\* @param {function} callBack 更新视图的回调函数*

*\*/*

function Watcher(*vm*, *key*, *callBack*) {

this.vm = vm;

this.key = key;

this.cb = callBack;

this.value = this.get(); *// 获取属性值*

}

Watcher.prototype = {

*/\* 收到通知执行更新方法 \*/*

update() {

this.run();

},

run() {

var value = this.vm.data[this.key]; *// 取得改变后的新值*

var oldValue = this.value; *// 旧值*

if (value !== oldValue) {

this.value = value; *// 更新旧值*

this.cb.call(this.vm, value); *// 执行更新视图函数*

}

},

get() {

Dep.target = this; *// 将自己赋值给Dep.target*

*// 执行Observe的get(),使得watcher添加到deps数组中*

var value = this.vm.data[this.key];

Dep.target = null; *// 添加成功后，释放*

return value;

}

}

此时，我们需要修改一下**监听器(Oberver)**中的get()方法：

get() {

if (Dep.target) { *// 如果是初始化的watcher，则添加到dep数组中*

dep.addSub(Dep.target); *// 添加watcher*

}

*/\**

*if(是否添加) {*

*// 添加计算属性...*

*}*

*\*/*

return value;

},

4.1到这里简易版的**订阅者(Watcher)**已经有模有样了，不过现在的监听器、发布者、订阅者都是独立的，要实现数据的双向绑定，我们需要使用**解析器(Compile)**将他们关联起来。解析器中也包含了对于模板指令的解析。为了解析Dom元素，我们需要处理节点，但又不能频繁的操作DOM，为此我们要使用[createDocumentFragment()](https://developer.mozilla.org/zh-CN/docs/Web/API/Document/createDocumentFragment)新建一个fragment片段，如何在虚拟的fragment上处理。(写在Compile.js中)

nodeToFragment(*el*) {

var child,

fragment = document.createDocumentFragment();*// 创建fragment*

while (child = el.firstChild) { *// 遍历节点*

fragment.appendChild(child); *// 创建fragment*

}

return fragment;

}

有了 fragment，我们可以对节点进行一一处理了。不过，我们需要一些小兵小将(所需的判断方法)来打头阵。

*/\* 1 === Element 元素 \*/*

*/\* 2 === Attr 属性 \*/*

*/\* 3 === Text 元素或属性中的文本 \*/*

*/\* 判断是否为文本 \*/*

isTextNode(*node*) {

return node.nodeType === 3;

},

*/\* 判断是否是节点 \*/*

isElementNode(*node*) {

return node.nodeType === 1

},

之后，便可以实现文本节点的处理了。

compileElement(*el*) {

var childNodes = el.childNodes; *//取得el节点的子节点合集*

var reg = /\{\{(.\*)\}\}/; *// 匹配 '{{ }}' 的正则表达式*

[].slice.call(childNodes).forEach(*node* => {

var text = node.textContent;

*/\* 判断是DOM节点还是文本节点 \*/*

if(this.isElementNode(node)) {

*// 进行指令解析*

}

else if (this.isTextNode(node) && reg.test(text)) {

*//exec() 方法用于检索字符串中的正则表达式的匹配。*

*//返回一个数组，其中存放匹配的结果。如果未找到匹配，则返回值为 null。*

this.compileText(node, reg.exec(text)[1].trim());

}

if (node.childNodes && node.childNodes.length) {

this.compileElement(node); *// 继续遍历子节点*

}

})

},

*/\* 文本节点订阅者初始化调用 \*/*

compileText(*node*, *key*) {

var self = this;

var initText = this.vm[key]; *// data: { exp : 'xxx' }*

this.updateText(node, initText);

new Watcher(this.vm, key, function (*value*) {

self.updateText(node, value);

})

},

*// 文本节点 更新视图/初始化 函数*

updateText(*node*, *value*) {

*// textContent 属性设置或返回指定节点的文本内容*

node.textContent = typeof value == 'undefined' ? '' : value;

},

获取到最外层的节点后，调用 compileElement() 方法，对所有子节点进行判断，如果节点是Dom节点进行指令解析(下文讲解)，如果是文本节点并且满足’{{ }}’的形式就进行编译和初始化视图(连接observer 和 watcher)。这样我们就能解析文本的节点了。

4.2 单单解析文本节点怎么能够呢，我们还要解析dom节点和其中的指令。所以我们需要一个解析指令的方法。

我们仍需要一些用来判断的方法

*/\* 判断是否是以 v- 开头 \*/*

isDirective(*attr*) {

return attr.indexOf('v-') === 0;

},

*/\* 判断是否是 v-on: 指令 \*/*

isEventDirective(*dir*) {

return dir.indexOf('on:') === 0;

}

然后是解析方法

*/\* 指令解析 \*/*

compile(*node*) {

var nodeAttrs = node.attributes; *//attributes 属性返回指定节点的属性集合，即 NamedNodeMap*

var self = this;

*//Array.prototype属性表示Array构造函数的原型，并允许为所有Array对象添加新的属性和方法。*

*//Array.prototype本身就是一个Array*

Array.prototype.forEach.call(nodeAttrs, function (*attr*) {

*//添加事件的方法名和前缀:*

*// v-on:click="onClick" ,则attrName = 'v-on:click' id="app" attrname= 'id'*

var attrName = attr.name;

if (self.isDirective(attrName)) { *//如果是指令：'v-'开头*

*//添加事件的方法名和前缀:v-on:click="onClick" ,exp = 'onClick'*

var exp = attr.value;

var dir = attrName.substring(2); *// 去除 'v-'*

if (self.isEventDirective(dir)) { *// 事件指令解析*

self.compileEvent(node, self.vm, exp, dir);

} else { *// model指令解析*

self.compileModel(node, self.vm, exp, dir);

}

node.removeAttribute(attrName); *// 解析完成，移除该属性*

}

})

},

*/\* model指令实现 \*/*

compileModel(*node*,*vm*,*exp*,*dir*) {

var self = this;

var val = this.vm[exp]; *// this.data[exp];*

this.modelUpdater(node,val);*// 更新函数*

new Watcher(this.vm,exp,function(*value*) {

self.modelUpdater(node,value);

});

*/\* 监听input的输入操作 \*/*

node.addEventListener('input',function(*e*) {

var newValue = e.target.value;*// 获取输入框的最新内容*

if(val === newValue) return;

self.vm[exp] = newValue; *// SelfVue.data[exp] = newValue*

val = newValue; *// 旧值更新*

})

},

*/\* v-on指令解析实现 \*/*

compileEvent(*node*, *vm*, *exp*, *dir*) {

*// 获取事件名, v-on:click => click*

let eventType = dir.split(':')[1];

let cb = vm.methods && vm.methods[exp]; *// 是否声明了methods*

if (eventType && cb) {

node.addEventListener(eventType, cb.bind(vm), false);

}

},

*// 主要是初始化*

modelUpdater(*node*, *value*) {

node.value = typeof value === 'undefined' ? '' : value;

}

我们有了：创建fragment(nodeToFragment)、遍历fragment节点(compileElement)、DOM节点指令解析方法(compile)、文本节点解析方法(compileText)、v-model解析方法(compileModel)、v-on解析方法(compileEvent)，现在我们要把这些功能模块整合起来，像Observer、Watcher那样。

*/\*\**

*\**

*\* @param {Element} el 需要绑定的DOM节点*

*\* @param {上下文环境} vm 指向SelfVue的作用域*

*\*/*

function Compile(*el*, *vm*) {

this.vm = vm; *//指向SelfVue的作用域*

this.el = document.querySelector(el); *// 绑定DOM节点*

this.fragment = null; *// 虚拟DOM节点*

this.init();

}

Compile.prototype = {

*/\* Compile初始化函数 \*/*

init() {

if (this.el) {

this.fragment = this.nodeToFragment(this.el); *//生成完整的虚拟DOM节点*

this.compileElement(this.fragment); *// 遍历fragment，对指定指令进行处理*

this.el.appendChild(this.fragment); *// 将处理完成的fragment渲染到页面*

} else {

alert("DOM元素不存在");

}

},

*/\* 将el节点的所有节点加入到fragment虚拟节点中 \*/*

nodeToFragment(*el*) {

*//创建节点对象(空的文档片段)*

let child, fragment = document.createDocumentFragment();

while (child = el.firstChild) {

fragment.appendChild(child);

}

return fragment;

},

*/\* 遍历fragment所有节点，对指定的节点进行特殊处理 \*/*

compileElement(*el*) {

let childNodes = el.childNodes;

let reg = /\{\{(.\*)\}\}/; *// 匹配的正则表达式*

*// [].slice.call 将伪数组转化为数组*

*// [].slice === Array.protoType.slice*

[].slice.call(childNodes).forEach(*node* => {

*// textContent 属性设置或返回指定节点的文本内容，以及它的所有后代(文本内容)*

let text = node.textContent;

if (this.isElementNode(node)) {

this.compile(node); *// 进行指令解析*

} else if (this.isTextNode(node) && reg.test(text)) {

*// exec()方法由于检索字符串中的正则表达式的匹配*

*// 返回一个数组，其中存放匹配的结果，如果未找到匹配，*

*// 则返回null*

this.compileText(node, reg.exec(text)[1].trim());

}

*// 如果还有子节点，则继续遍历*

if (node.childNodes && node.childNodes.length) {

this.compileElement(node);

}

})

},

*/\* 指令解析 \*/*

compile(*node*) {

let nodeAttrs = node.attributes; *// attributes 属性返回指定DOM节点的属性合集*

let self = this;

*//Array.prototype属性表示Array构造函数的原型，并允许为所有Array对象添加新的属性和方法。*

*//Array.prototype本身就是一个Array*

Array.prototype.forEach.call(nodeAttrs, function (*attr*) {

*// 添加事件的方法名和前缀*

*// v-on:click="func",则attrName = 'v-on:click'*

let attrName = attr.name;

if (self.isDirective(attrName)) { *// 是以 'v-' 开头*

*//添加事件的方法值*

*// v-on:click = 'func',则 exp = 'func'*

let exp = attr.value;

let dir = attrName.substring(2); *// 去除 'v-'*

if (self.isEventDirective(dir)) { *// 事件指令解析*

self.compileEvent(node, self.vm, exp, dir);

} else { *// model指令解析*

self.compileModel(node, self.vm, exp, dir);

}

node.removeAttribute(attrName); *// 移除指令属性*

}

})

},

*/\* model指令解析实现 \*/*

compileModel(*node*, *vm*, *exp*, *dir*) {

let self = this;

let value = this.vm[exp]; *// 等价于 this.data[exp]*

this.modelUpdater(node, value); *// 初始化视图(input框默认值)*

*// 是否可去掉*

new Watcher(this.vm, exp, function (*value*) {

self.modelUpdater(node, value);

});

*// 监听input的输入操作*

node.addEventListener('input', function (*e*) {

let newValue = e.target.value; *// 获取输入的最新内容*

if (value === newValue) return;

self.vm[exp] = newValue; *// 更新SelfVue.data[exp]的值*

value = newValue; *// 旧值更新*

})

},

*/\* model 初始化/更新视图函数 \*/*

*// 主要是初始化，更新视图功能重复了*

modelUpdater(*node*, *value*) {

node.value = typeof value === 'undefined' ? '' : value;

},

*/\* v-on指令解析实现 \*/*

compileEvent(*node*, *vm*, *exp*, *dir*) {

*// 获取事件名, v-on:click => click*

let eventType = dir.split(':')[1];

let cb = vm.methods && vm.methods[exp]; *// 是否声明了methods*

if (eventType && cb) {

node.addEventListener(eventType, cb.bind(vm), false);

}

},

*/\* 文本属性解析 \*/*

compileText(*node*, *key*) {

let self = this;

let initText = this.vm[key]; *// 等价于 SelfVue.data[key]*

this.updateText(node, initText); *//文本节点初始化视图*

new Watcher(this.vm, key, function (*value*) {

self.updateText(node, value);

})

},

*/\* 文本节点 初始化/更新视图函数 \*/*

updateText(*node*, *value*) {

node.textContent = typeof value === 'undefined' ? '' : value;

},

*/\* 判断是否是 v-on: 指令 \*/*

isEventDirective(*dir*) {

return dir.indexOf('on:') === 0;

},

*/\*\**

*\* nodeType === 1 : DOM节点*

*\* nodeType === 2 : 属性(attr属性)节点*

*\* nodeType === 3 : 文本节点*

*\*/*

*/\* 判断是否是DOM节点 \*/*

isElementNode(*node*) {

return node.nodeType === 1;

},

*/\* 判断是否是文本节点 \*/*

isTextNode(*node*) {

return node.nodeType === 3;

},

*/\* 判断是否是以 v- 开头 \*/*

isDirective(*attr*) {

return attr.indexOf('v-') === 0;

}

}

一路走来，我们终于有了**监听器(Observer)**、**发布者(Dep)**、**订阅者(Watcher)**、**解析器(Compile)**，现在通过SelfVue方法把他们联系起来(如同new Vue)。(写在SelfVue中)

*/\*\**

*\**

*\* @param {Object} object 需要绑定的数据*

*\*/*

function SelfVue(*object*) {

this.vm = this; *// 绑定自身作用域*

this.data = object.data; *// 绑定数据*

this.methods = object.methods; *//绑定方法*

observe(this.data); *// 遍历数据，是每个属性为响应式*

new Compile(object.el, this.vm); *//绑定页面DOM元素，并且解析指令*

return this;

}

可以看到，SelfVue() 方法是入口函数。连接了Observser和Compile (Compile连接了Watcher)，这样我们就连接了所有的参与者。我们可以来试一下。

<!DOCTYPE html>

<head>

<script src="Observe.js"></script>

<script src="watcher.js"></script>

<script src="Compile.js"></script>

<script src="SelfVue.js"></script>

</head>

<body>

<div id="demo">

双向绑定：<h1>{{title}}</h1>

双向绑定：<input type="text" v-model="title">

</div>

<script>

var sv = new SelfVue({

el: '#demo',

data: {

title: "Hello SprWu",

end: ''

}

});

window.setTimeout(() => {

sv.title = "SprWu SprWu";

}, 3000);

</script>

</body>

</html>

打开浏览器我们一看，为什么没有我们想要的效果？而且浏览器也没有报错！其实看代码，我们的数据是放在 sv实例的data 上的，而我们访问和修改是 sv.title = "SprWu SprWu"，这样当然没效果啦。那怎么办呢？我不想再从头慢慢改下来，而且sv.data.title

写着好麻烦。没关系，我们在SelfVue()上挂一个属性访问代理。

*/\*\**

*\**

*\* @param {Object} object 需要绑定的数据*

*\*/*

function SelfVue(*object*) {

this.vm = this; *// 绑定自身作用域*

this.data = object.data; *// 绑定数据*

this.methods = object.methods; *//绑定方法*

*// 属性访问代理,使得 实例.属性名 = 实例.data.属性名*

Object.keys(this.data).forEach( *key* => this.proxykeys(key));

observe(this.data); *// 遍历数据，是每个属性为响应式*

new Compile(object.el, this.vm); *//绑定页面DOM元素，并且解析指令*

return this;

}

SelfVue.prototype = {

*/\* 属性访问代理 \*/*

proxykeys(*key*) {

Object.defineProperty(this, key, {

enumerable: false, *// 不可被遍历*

configurable: true, *// 可修改*

get() {

*// 此get()，只在页面初始化时调用*

return this.data[key];

},

set(*newValue*) {

*// 实现属性访问代理*

this.data[key] = newValue;

}

})

}

}

再打开浏览器，已经有效果了。好了，我们完成了数据的双向绑定。到现在，是不是对于Vue的”数据驱动”有了一些些理解。他帮助我们可以快速的搭建页面，实现动态变化而不需要去频繁的操作DOM。

接下来，我们在此基础上添加一个功能-------计算属性(本文所展示的计算属性功能，其思路与设计代码可能与Vue官方或其他作者不同，也可能有许多不足或错误之处)。

\* 绑定的属性、依赖的属性、处理数据的函数

\* 拿到依赖的数据并执行处理数据的函数，初始化页面

\* 把处理数据的回调函数存在**监听器(Observser)** 中

\* 当依赖属性改变时，触发**监听器(Observer)**并执行该回调函数更新页面

因为计算属性需要依赖 sv.data 中至少一个属性，而且返回值是经过处理的。所以我们需要绑定值和一个处理数据的回调函数：

*/\*\**

*\**

*\* @param {Object} obj 绑定计算属性的属性*

*\* @param {string} key 键名，属性名*

*\* @param {function} comFunc 计算属性的计算函数*

*\*/*

function Computed(*obj*, *key*, *comFunc*) {

*// 绑定调用函数的对象(可优化)*

comFUnc = comFunc.bind(sv.data);

let value = comFunc(); *// 初始化计算结果*

sv.data.say = value; *// 初始化*

}

我们把调用回调函数*comFunc* 的对象绑定为 sv.data，并调用一次将值给value用以初始化页面。那怎么让回调函数加入**监听器(Observer)**中呢？ 和**订阅者(Watcher)**一样，让回调函数赋值到一个变量上，再在get()里做个判断。

更改后的Computed.js。

*/\*\**

*\**

*\* @param {Object} obj 绑定计算属性的属性*

*\* @param {string} key 键名，属性名*

*\* @param {function} comFunc 计算属性的计算函数*

*\*/*

function Computed(*obj*, *key*, *comFunc*) {

*// 绑定调用函数的对象(需优化)*

comFUnc = comFunc.bind(sv.data);

*/\* 更新视图函数 \*/*

ComState = function() {

sv.data.say = comFunc();*// 更新内容(触发Observe的set() => Watcher的回调函数)*

};

let value = comFunc(); *// 初始化计算结果*

sv.data.say = value; *// 初始化*

ComState = null; *// 加入deps数组成功后释放*

}

更改后的Obersver.js。

*/\*\**

*\**

*\* @param {object} data 写在实例上的数据 \*.data: {x1:xxx,x2:xxx}*

*\* @param {string} key 键名,属性名*

*\* @param {\*} value 键值，属性值*

*\*/*

function defineReactive(*data*, *key*, *value*) {

observe(value); *// 递归遍历*

let dep = new Dep(); *//消息订阅器,保存订阅者*

let deps = []; *// 保存所有的计算属性回调*

Object.defineProperty(data, key, {

enumerable: true, *// 可遍历(for..of,for..in ...)*

configurable: true, *// 可编辑*

get() {

if(Dep.target) { *// 如果是初始化的 watcher,则加入dep数组*

dep.addSub(Dep.target);

}

if(ComState) { *// 如果是初始化的 compile,则加入deps数组*

deps.push(ComState);

}

return value;

},

set(*newValue*) {

*// 值未发生改变，则不执行*

if(value === newValue) return;

value = newValue;

deps.forEach( *func* => func()); *// 执行计算属性的回调函数，重新计算属性值*

dep.notify(); *// 通知所有订阅者更新函数*

}

})

}

ComState 上挂载了一个回调函数，用以赋值并更新数据。当执行value = comFunc()时，comFunc内获取依赖数据值的操作就会触发其get()，也就能把ComState加入到deps数组中，添加成功后释放ComState。当依赖数据改变时，触发了set()，就循环deps数组，并执行所有依赖此属性的计算属性的回调函数。

我们整合所有功能来试试看。

<!DOCTYPE html>

<head>

<title>数据双向绑定及计算属性</title>

<script src="Observe.js"></script>

<script src="Watcher.js"></script>

<script src="Compile.js"></script>

<script src="SelfVue.js"></script>

<script src="Computed.js"></script>

</head>

<body>

<div id="demo">

双向绑定： <h1>{{name}}</h1>

双向绑定： <input type="text" v-model="name">

<hr>

计算属性(v-model)： <h1 id="cmp">{{say}}</h1>

<hr>

指令(v-on)： <input type="button" v-on:click="changeColor" value="改变计算属性的颜色">

</div>

<script>

var sv = new SelfVue({

el: '#demo',

data: {

name: 'SprWu',

say: ''

},

methods: {

changeColor() {

document.querySelector('#cmp').style.color = 'deepskyblue';

}

}

});

var obj = {

say: ''

}

Computed(obj, 'say', function() {

return `Hello ${sv.name} !`;

})

window.setTimeout(()=> {

sv.name = 'SPRWU'

},2000);

</script>

</body>

</html>



完整代码： <https://gitee.com/supme/computed_for_vue>