#### 前言：

对于传统的dom操作，当数据变化时更新视图需要先获取到目标节点，然后将改变后的值放入节点中，视图发生变化时，需要绑定事件修改数据。双向数据恰好能解决这种复杂的操作，当数据发生变化时会自动更新视图，视图发生变化时也会自动更新数据，极大的提高了开发效率。那双向数据绑定到底是怎么实现的了，下面来讲述双向数据绑定的原理。

#### 1、Vue双向数据绑定的原理。

Vue实现双向数据绑定是采用数据劫持和发布者-订阅者模式。数据劫持是利用ES5的Object.defineProperty(obj,key,val)方法来劫持每个属性的getter和setter，在数据变动是发布消息给订阅者，从而触发相应的回调来更新视图，下面来一步步实现。

```

<div id="app">

用户名：<input type="text" v-model="name">

密码：<input type="text" v-model="passWord">

{{name}} {{passWord}}

<div><div>{{name}}</div></div>

</div>

<script>

function Vue(option){

this.data = option.data;

this.id = option.el;

var dom = nodeToFragment(document.getElementById(this.id), this);

document.getElementById(this.id).appendChild(dom);

}

var vm = new Vue({

el: "app",

data: {

name: "zhangsan",

passWord: "123456"

}

})

<script>

```

如上一段html，想要实现双向数据绑定，我们需要先解析这一段html，找到带有v-model指令和{{}}的节点（此处节点包括元素节点和文本节点）,然后我们定义了一个Vue的构造函数，在实例化创建对象vm时，传入id='app'和对应的数据data，我们现在需要实现的功能是，当实例化创建对象时，将对应的'name'和'passWord'属性渲染到页面上。

在解析上面一段模板时，需要先了解一下DocuemntFragment（碎片化文档）这个概念，你可以把他认为是一个dom节点收容器，当你创造了10个节点，当每个节点都插入到文档当中都会引发一次浏览器的回流，也就是说浏览器要回流10次，十分消耗资源。而使用碎片化文档，也就是说我把10个节点都先放入到一个容器当中，最后我再把容器直接插入到文档就可以了！浏览器只回流了1次。

解析html

```

// 此方法是将传入的dom节点转为文档碎片,参数node是需要解析的html片段，vm是Vue构造函数实例化的对象。

function nodeToFragment(node, vm){

// 创建一个文档碎片

var fragment = document.createDocumentFragment();

var child;

// 获取到node中的第一个子节点，当有子节点时，执行循环体

while(child = node.firstChild){

// appendChild会将参数中节点移除，因此此循环会将node中的节点一个个移除，移动到fragment文档碎片中，直到node中没有节点，循环结束。

fragment.appendChild(child);

}

// 此处fragment已经获取到node中所有节点，loopNode函数用来循环每一层的节点。

loopNode(fragment.childNodes, vm);

return fragment;

}

function loopNode(nodes, vm){

//此处传入的nodes是一个类数组，使用Array.from()方法将其转化为数组。

Array.from(nodes).forEach((node) => {

// 此处得到的node是nodes中的直接子节点，compile函数是用来解析这些节点，如果是元素节点，解析是否有v-model指令，如果是文本节点，解析是否有{{}}。

compile(node, vm);

// 如果node还有子节点，则继续解析

if(node.childNodes.length>0){

loopNode(node.childNodes, vm);

}

})

}

function compile(node, vm){

// 如果是元素节点

if(node.nodeType === 1){

// 获得元素节点上所有的属性，以键值对的方式存储在attrs中，attrs属于类数组

var attrs = node.attributes;

Array.from(attrs).forEach(element => {

if(element.nodeName == "v-model"){

var name = element.nodeValue;

// 初始化带有v-model指令的元素的值

node.value = vm.data[name];

}

});

}

// 正则匹配到文本中有{{}}的文本

var reg = /\{\{([^}]\*)\}\}/g;

var textContent = node.textContent;

// 如果是文本节点且文本中带有{{}}的节点

if(node.nodeType === 3 && reg.test(textContent)){

// 将文本内容存放在当前节点的自定义属性上

node.my = textContent;

// 此处node.textContent 和 node.my的文本一样，如果上一步不将文本存储到自定义属性中，那么下次将无法匹配到{{}}，replace方法用来替换文本中的{{name}}和{{passWord}}。

node.textContent = node.my.replace(reg, function(){

var attr = arguments[0].slice(2,arguments[0].length-2);

return vm.data[attr];

})

}

}

```

上面我们已经实现了将data中的属性填充到页面中，接下来我们需要做的是，当data中属性值发生变化时，我们需要监听到数据的变化，Vue中对数据监听使用的是Object.defineProperty(data,key,val)方法(不清楚该方法的可[查阅](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Reference/Global\_Objects/Object/defineProperty))，Object.defineProperty(data,key,val)可以监听对象属性的变化，当获取data中某个属性的值时，会调用该属性的get()方法，当修改某个属性的值时会调用当前属性的set()方法。

```

function observe(data){

if(typeof data != 'object' || !data){

return

}

Object.keys(data).forEach((key)=>{

defineReactive(data, key, data[key]);

})

}

function defineReactive(data, key, val){

// data中子属性是对象时，继续监听

observe(val);

Object.defineProperty(data, key, {

get: function(){

return val;

},

set: function(newVal){

if(newVal !== val){

val = newVal;

} else {

return;

}

}

})

}

```

修改Vue构造函数如下，当实例化Vue时，实现了对数据data的监听，并解析模板，将data中对应属性填充到页面中。

![image](/img/bVbLXqr)

但是，当data中属性值发生变化时，页面并不会更新，那接下来我们需要解决的就是，当data中属性发生变化时，自动更新视图，视图发生变化时，主动更新数据，连接视图和数据我们需要在定义一个构造函数Watcher。

首先我们来考虑下当data中name属性发生变化时，我们需要更新的视图有如下三个节点，一个元素节点和两个文本节点

![image](/img/bVbLXDs)

当data中passWord属性变化时，需要更新的视图有两个节点

![image](/img/bVbLXE1)

也就是说，当某个属性发生变化时，我们可能要更新多个视图，那我们如何去定位需要更新那些节点了？因此我们需要将绑定了data中属性的节点保存到一个数组中，当data中对应属性发生变化时，循环数组，拿到节点，执行更新方法。

回顾一下我们compile中的代码，下图中标记1、2处就是获取到data中的属性名。接下来我们定义一个Watcher构造函数，在解析模板时实例化Watcher，3、4处是新增代码，实例化Watcher。

![image](/img/bVbLXWO)

Watcher构造函数中有两个方法，一个update方法和一个get方法，实例化Watcher时调用Watcher中的get方法，此方法会触发data中对应的属性的get方法。

```

function Watcher(vm, node, name){

this.vm = vm;

this.node = node;

this.name = name;

this.value = this.get();

}

Watcher.prototype.get = function(){

//触发data中属性get方法前将当前实例化对象存入target属性中

Dep.target = this;

//取data中的this.name属性，会触发该属性的get方法

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

Dep.target = null;

return value;

}

Watcher.prototype.update = function(){

}

```

上文中提到，我们需要定义数组来存储对应属性的节点，也就是说，data中每个属性都必须有一个数组来存储节点，下面我们来定义一个Dep构造函数，用来收集节点。

```

function Dep(){

// 存放Watcher的实例对象

this.subs = [];

}

Dep.prototype.addSub = function(sub){

this.subs.push(sub);

}

Dep.prototype.notify = function(){

this.subs.forEach((sub)=>{

sub.update();

})

}

```

每个属性都需要一个数组，因此我们在监听data属性时实例化Dep，Dep的实例在闭包的情况下创建，我们可以修改数据监听中的get方法，上文在实例化Watcher时，触发get方法，将Watcher的实例存入数组中，当修改data中属性值时，调用set方法，Dep实例对象调用notify方法，实现更新。

```

//修改后的defineReactive方法

function defineReactive(data, key, val){

//为每个属性创建一个Dep实例

var dep = new Dep();

observe(val);

Object.defineProperty(data, key, {

get: function(){

//实例化Watcher时，触发了get方法，此时Dep.target为Watcher实例化对象

Dep.target && dep.addSub(Dep.target);

return val;

},

set: function(newVal){

if(newVal !== val){

val = newVal;

// 当调用set方法时，通知所有订阅者执行更新方法

dep.notify();

} else {

return;

}

}

})

}

```

实现更新方法

```

function Watcher(vm, node, name){

...

}

Watcher.prototype.get = function(){

...

}

Watcher.prototype.update = function(){

if(this.node.nodeType === 1){

this.node.nodeValue = this.get();

} else {

this.node.textContent = this.node.my.replace(/\{\{([^}]\*)\}\}/g, function(){

var attr = arguments[0].slice(2,arguments[0].length-2);

return this.vm.data[attr];

})

}

}

```

完成到这里，我们就已经实现了数据变化时自动更新视图，我们来梳理一下流程。就拿上面例子来说，当我们执行vm.data['name'] = 'lisi'时，便会触发set方法，set方法中调用Dep实例的notify方法，此方法会遍历this.subs数组，这个数组中存放的元素是Watcher的实例化对象，调用sub.update()方法便会更新视图。

当视图发生变化时，需要修改相应数据，只需要给相应节点绑定事件即可，修改compile方法如下，给相应节点增加input事件。

```

if(node.nodeType === 1){

// 获得元素节点上所有的属性，以键值对的方式存储在attr中，attr属于类数组

var attr = node.attributes;

Array.from(attr).forEach(element => {

if(element.nodeName == "v-model"){

var name = element.nodeValue;

// 给带有v-model指令的元素绑定input事件

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

vm.data[name] = e.target.value;

})

// 初始化带有v-model指令的元素的值

node.value = vm.data[name];

new Watcher(vm, node, name);

}

});

}

```

到这里双向数据绑定就完成了，下面附上完整代码。

```

<!DOCTYPE html>

<html lang="en">

<head>

<meta charset="UTF-8">

<meta name="viewport" content="width=device-width, initial-scale=1.0">

<meta http-equiv="X-UA-Compatible" content="ie=edge">

<title>Vue双向数据绑定</title>

</head>

<body>

<div id="app">

用户名：<input type="text" v-model="name">

密码：<input type="text" v-model="passWord">

{{name}} {{passWord}}

<div>

<div>{{name}}</div>

</div>

</div>

<script>

function Vue(option) {

this.data = option.data;

this.id = option.el;

observe(this.data);

var dom = nodeToFragment(document.getElementById(this.id), this);

document.getElementById(this.id).appendChild(dom);

}

function nodeToFragment(node, vm) {

// 创建一个文档碎片

var fragment = document.createDocumentFragment();

var child;

// 获取到node中的第一个节点

while (child = node.firstChild) {

// appendChild会将传入的节点移除，因此此循环会将node中的节点一个个移除，移动到fragment文档碎片中。

fragment.appendChild(child);

}

// console.dir(fragment);

loopNode(fragment.childNodes, vm);

return fragment;

}

function loopNode(nodes, vm) {

//此处传入的nodes是一个类数组，将其转化为数组

Array.from(nodes).forEach((node) => {

compile(node, vm);

// 如果node还有子节点，则继续解析

if (node.childNodes.length > 0) {

loopNode(node.childNodes, vm);

}

})

}

function compile(node, vm) {

// 如果是元素节点

if (node.nodeType === 1) {

// 获得元素节点上所有的属性，以键值对的方式存储在attr中，attr属于类数组

var attr = node.attributes;

Array.from(attr).forEach(element => {

if (element.nodeName == "v-model") {

var name = element.nodeValue;

// 给带有v-model指令的元素绑定input时间

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

vm.data[name] = e.target.value;

})

// 初始化带有v-model指令的元素的值

node.value = vm.data[name];

new Watcher(vm, node, name);

}

});

}

// 正则匹配到文本中有{{}}的文本

var reg = /\{\{([^}]\*)\}\}/g;

var textContent = node.textContent;

// 如果是文本节点且文本中带有{{}}的节点

if (node.nodeType === 3 && reg.test(textContent)) {

// 将文本内容存放在当前节点的自定义属性上

node.my = textContent;

// 此处node.textContent 和 node.my的文本一样，如果上一步不将文本存储到自定义属性中，那么下次将无法匹配到{{}}。

node.textContent = node.my.replace(reg, function () {

var attr = arguments[0].slice(2, arguments[0].length - 2);

new Watcher(vm, node, attr);

return vm.data[attr];

})

}

}

function observe(data) {

if (typeof data != 'object' || !data) {

return

}

Object.keys(data).forEach((key) => {

defineReactive(data, key, data[key]);

})

}

function defineReactive(data, key, val) {

var dep = new Dep();

observe(val);

Object.defineProperty(data, key, {

get: function () {

Dep.target && dep.addSub(Dep.target);

return val;

},

set: function (newVal) {

if (newVal !== val) {

val = newVal;

dep.notify();

} else {

return;

}

}

})

}

function Dep() {

this.subs = [];

}

Dep.prototype.addSub = function (sub) {

this.subs.push(sub);

}

Dep.prototype.notify = function () {

this.subs.forEach((sub) => {

sub.update();

})

}

function Watcher(vm, node, name) {

this.vm = vm;

this.node = node;

this.name = name;

this.value = this.get();

}

Watcher.prototype.get = function () {

Dep.target = this;

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

Dep.target = null;

return value;

}

Watcher.prototype.update = function () {

if (this.node.nodeType === 1) {

this.node.nodeValue = this.get();

} else {

this.node.textContent = this.node.my.replace(/\{\{([^}]\*)\}\}/g, function () {

var attr = arguments[0].slice(2, arguments[0].length - 2);

return this.vm.data[attr];

})

}

}

var vm = new Vue({

el: "app",

data: {

name: "lishibo",

passWord: "123456",

obj: {

obj1: 'obj1'

},

arr: ['arr1', 'arr2']

}

})

</script>

</body>

</html>

```