https://www.jb51.net/article/124439.htm

http://www.php.cn/js-tutorial-393678.html

https://www.jianshu.com/p/1032ecd62b3a

https://www.cnblogs.com/leaf930814/p/9014200.html

http://www.php.cn/js-tutorial-393678.html

https://blog.csdn.net/it\_rod/article/details/79516578

**通过源码分析Vue的双向数据绑定详解**

**前言**

虽然工作中一直使用Vue作为基础库，但是对于其实现机理仅限于道听途说，这样对长期的技术发展很不利。所以最近攻读了其源码的一部分，先把双向数据绑定这一块的内容给整理一下，也算是一种学习的反刍。

本篇文章的Vue源码版本为v2.2.0开发版。

Vue源码的整体架构无非是初始化Vue对象，挂载数据data/props等，在不同的时期触发不同的事件钩子，如created() / mounted() / update()等，后面专门整理各个模块的文章。这里先讲双向数据绑定的部分，也是最主要的部分。

**设计思想：观察者模式**

Vue的双向数据绑定的设计思想为观察者模式，为了方便，下文中将被观察的对象称为观察者，将观察者对象触发更新的称为订阅者。主要涉及到的概念有：

1、Dep对象：Dependency依赖的简写，包含有三个主要属性id, subs, target和四个主要函数addSub, removeSub, depend, notify，是观察者的依赖集合，负责在数据发生改变时，使用notify()触发保存在subs下的订阅列表，依次更新数据和DOM。

1. id: 每个观察者(依赖对象)的唯一标识。
2. subs: 观察者对象的订阅者列表。
3. target: 全局唯一的订阅者对象，因为只能同时计算和更新一个订阅者的值。
4. addSub(): 使用`push()`方法添加一个订阅者。
5. removeSub(): 使用`splice()`方法移除一个订阅者。
6. depend(): 将自己添加到当前订阅者对象的依赖列表。
7. notify(): 在数据被更新时，会遍历subs对象，触发每一个订阅者的更新。

2、Observer对象：即观察者，包含两个主要属性value, dep。做法是使用getter/setter方法覆盖默认的取值和赋值操作，将对象封装为响应式对象，每一次调用时更新依赖列表，更新值时触发订阅者。绑定在对象的\_\_ob\_\_原型链属性上。

1. value: 原始值。
2. dep: 依赖列表。

**源码实战解析**

有过Vue开发基础的应该都了解其怎么初始化一个Vue对象：

[?](https://www.jb51.net/article/124439.htm)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | new Vue({   el: '#container',   data: {    count: 100   },   ...  }); |

那么我们就从这个count说起，看它是怎么完成双向数据绑定的。

下面的代码片段中英文注释为尤雨溪所写，中文注释为我所写，英文注释更能代表开发者的清晰思路。

首先从全局的初始化函数调用：initMixin(Vue$3); ，这里的Vue$3对象就是全局的Vue对象，在此之前已经挂载了Vue的各种基本数据和函数。这个函数体就是初始化我们上面声明Vue语句的过程化逻辑，取主体代码来看：

[?](https://www.jb51.net/article/124439.htm)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15 | // 这里的options就是上面声明Vue对象的json对象  Vue.prototype.\_init = function (options) {   ...   var vm = this;   ...   initLifecycle(vm);   initEvents(vm);   initRender(vm);   callHook(vm, 'beforeCreate');   // 这里就是我们接下来要跟进的初始化Vue参数   initState(vm);   initInjections(vm);   callHook(vm, 'created');   ...   }; |

这里主要完成了初始化事件、渲染、参数、注入等过程，并不断调用事件钩子的回调函数。下面来到如何初始化参数：

[?](https://www.jb51.net/article/124439.htm)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | function initState (vm) {   vm.\_watchers = [];   var opts = vm.$options;   if (opts.props) { initProps(vm, opts.props); }   if (opts.methods) { initMethods(vm, opts.methods); }   // 我们的count在这里初始化   if (opts.data) {   initData(vm);   } else {   observe(vm.\_data = {}, true /\* asRootData \*/);   }   if (opts.computed) { initComputed(vm, opts.computed); }   if (opts.watch) { initWatch(vm, opts.watch); }  } |

这里依次检测参数中包含的props/methods/data/computed/watch并进入不同的函数进行初始化，这里我们只关心initData：

[?](https://www.jb51.net/article/124439.htm)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | function initData (vm) {   var data = vm.$options.data;   data = vm.\_data = typeof data === 'function'   ? data.call(vm)   : data || {};   if (!isPlainObject(data)) {   data = {};   }   ...   // observe data   observe(data, true /\* asRootData \*/); |

可以看到Vue的data参数支持对象和回调函数，但最终返回的一定是对象，否则使用空对象。接下来就是重头戏了，我们如何将data参数设置为响应式的：

[?](https://www.jb51.net/article/124439.htm)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27 | /\*\*   \* Attempt to create an observer instance for a value,   \* returns the new observer if successfully observed,   \* or the existing observer if the value already has one.   \*/  function observe (value, asRootData) {   if (!isObject(value)) {   return   }   var ob;   if (hasOwn(value, '\_\_ob\_\_') && value.\_\_ob\_\_ instanceof Observer) {   ob = value.\_\_ob\_\_;   } else if (   /\* 为了防止value不是单纯的对象而是Regexp或者函数之类的，或者是vm实例再或者是不可扩展的 \*/   observerState.shouldConvert &&   !isServerRendering() &&   (Array.isArray(value) || isPlainObject(value)) &&   Object.isExtensible(value) &&   !value.\_isVue   ) {   ob = new Observer(value);   }   if (asRootData && ob) {   ob.vmCount++;   }   return ob  } |

这里的英文注释非常清晰，就是为了给该对象新建一个观察者类，如果存在则返回已存在的（比如互相引用或依赖重复），可以看到这个观察者列表放置在对象的\_\_ob\_\_属性下。下面我们看下这个Observer观察者类：

[?](https://www.jb51.net/article/124439.htm)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24 | /\*\*   \* Observer class that are attached to each observed   \* object. Once attached, the observer converts target   \* object's property keys into getter/setters that   \* collect dependencies and dispatches updates.   \*/  var Observer = function Observer (value) {   this.value = value;   this.dep = new Dep();   this.vmCount = 0;   // def函数是defineProperty的简单封装   def(value, '\_\_ob\_\_', this);   if (Array.isArray(value)) {   // 在es5及更低版本的js里，无法完美继承数组，这里检测并选取合适的函数   // protoAugment函数使用原型链继承，copyAugment函数使用原型链定义（即对每个数组defineProperty）   var augment = hasProto    ? protoAugment    : copyAugment;   augment(value, arrayMethods, arrayKeys);   this.observeArray(value);   } else {   this.walk(value);   }  }; |

在Observer类的注释里也清楚的说明，它会被关联到每一个被检测的对象，使用getter/setter修改其默认读写，用于收集依赖和发布更新。其中出现了三个我们需要关心的东西Dep类/observeArray/walk，我们先看observeArray的源码：

[?](https://www.jb51.net/article/124439.htm)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | /\*\*   \* Observe a list of Array items.   \*/  Observer.prototype.observeArray = function observeArray (items) {   for (var i = 0, l = items.length; i < l; i++) {   observe(items[i]);   }  }; |

它不过是在Observer类和observe方法中间的一层递归，因为我们观察的只能是对象，而不能是数字、字符串或者数组（数组的观察比较特殊，事实上是重构了方法来触发更新，后面会讲到）。那我们接下来看下Dep类是做什么用的：

[?](https://www.jb51.net/article/124439.htm)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | /\*\*   \* A dep is an observable that can have multiple   \* directives subscribing to it.   \*/  var Dep = function Dep () {   this.id = uid$1++;   this.subs = [];  }; |

注释里告诉我们Dep类是一个会被多个指令订阅的可被观察的对象，这里的指令就是我们在html代码里书写的东西，如:class={active: hasActive}或{{ count }} {{ count \* price }} ，而他们就会订阅hasActive/count/price这些对象，而这些订阅他们的对象就会被放置在Dep.subs列表中。每一次新建Dep对象，就会全局uid递增，然后传给该Dep对象，保证唯一性id。

我们接着看刚才的walk函数做了什么：

[?](https://www.jb51.net/article/124439.htm)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | /\*\*   \* Walk through each property and convert them into   \* getter/setters. This method should only be called when   \* value type is Object.   \*/  Observer.prototype.walk = function walk (obj) {   var keys = Object.keys(obj);   for (var i = 0; i < keys.length; i++) {   defineReactive$$1(obj, keys[i], obj[keys[i]]);   }  }; |

看来和名字一样，它只是走了一遍，那我们来看下defineReactive$$1做了什么：

[?](https://www.jb51.net/article/124439.htm)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48 | /\*\*   \* Define a reactive property on an Object.   \*/  function defineReactive$$1 (obj, key, val, customSetter) {   var dep = new Dep();     var property = Object.getOwnPropertyDescriptor(obj, key);   if (property && property.configurable === false) {   return   }     // cater for pre-defined getter/setters   var getter = property && property.get;   var setter = property && property.set;     var childOb = observe(val);   Object.defineProperty(obj, key, {   enumerable: true,   configurable: true,   get: function reactiveGetter () {    var value = getter ? getter.call(obj) : val;    if (Dep.target) {    dep.depend();    if (childOb) {     childOb.dep.depend();    }    if (Array.isArray(value)) {     dependArray(value);    }    }    return value   },   set: function reactiveSetter (newVal) {    var value = getter ? getter.call(obj) : val;    // 脏检查，排除了NaN !== NaN的影响    if (newVal === value || (newVal !== newVal && value !== value)) {    return    }    if (setter) {    setter.call(obj, newVal);    } else {    val = newVal;    }    childOb = observe(newVal);    dep.notify();   }   });  } |

终于找到重头戏了，这里真正使用了getter/setter代理了对象的默认读写。我们首先新建一个Dep对象，利用闭包准备收集依赖，然后我们使用observe观察该对象，注意此时与上面相比少了一个asRootData = true的参数。

我们先来看取值的代理get，这里用到了Dep.target属性和depend()方法，我们来看看它是做什么的：

[?](https://www.jb51.net/article/124439.htm)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18 | // the current target watcher being evaluated.  // this is globally unique because there could be only one  // watcher being evaluated at any time.  Dep.target = null;    Dep.prototype.depend = function depend () {   if (Dep.target) {   Dep.target.addDep(this);   }  };    Dep.prototype.notify = function notify () {   // stablize the subscriber list first   var subs = this.subs.slice();   for (var i = 0, l = subs.length; i < l; i++) {   subs[i].update();   }  }; |

注释看的出来Dep.target是全局唯一的watcher对象，也就是当前正在指令计算的订阅者，它会在计算时赋值成一个watcher对象，计算完成后赋值为null。而depend是用于对该订阅者添加依赖，告诉它你的值依赖于我，每次更新时应该来找我。另外还有notify()的函数，用于遍历所有的依赖，通知他们更新数据。

这里多看一下addDep()的源码：

[?](https://www.jb51.net/article/124439.htm)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | /\*\*   \* Add a dependency to this directive.   \*/  Watcher.prototype.addDep = function addDep (dep) {   var id = dep.id;   if (!this.newDepIds.has(id)) {   this.newDepIds.add(id);   this.newDeps.push(dep);   if (!this.depIds.has(id)) {    // 使用push()方法添加一个订阅者    dep.addSub(this);   }   }  }; |

可以看到它有去重的机制，当重复依赖时保证相同ID的依赖只有一个。订阅者包含3个属性newDepIds/newDeps/depIds分别存储依赖信息，如果之前就有了这个依赖，那么反过来将该订阅者加入到这个依赖关系中去。

接着看get方法中的dependArray() ：

[?](https://www.jb51.net/article/124439.htm)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13 | /\*\*   \* Collect dependencies on array elements when the array is touched, since   \* we cannot intercept array element access like property getters.   \*/  function dependArray (value) {   for (var e = (void 0), i = 0, l = value.length; i < l; i++) {   e = value[i];   e && e.\_\_ob\_\_ && e.\_\_ob\_\_.dep.depend();   if (Array.isArray(e)) {    dependArray(e);   }   }  } |

可以看到我们不能像对象一样监听数组的变化，所以如果获取一个数组的值，那么就需要将数组中所有的对象的观察者列表都加入到依赖中去。

这样get方法读取值就代理完成了，接下来我们看set方法代理赋值的实现，我们先获取原始值，然后与新赋的值进行比较，也叫脏检查，如果数据发生了改变，则对该数据进行重新建立观察者，并通知所有的订阅者更新。

接下来我们看下数组的更新检测是如何实现的：

[?](https://www.jb51.net/article/124439.htm)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38 | /\*   \* not type checking this file because flow doesn't play well with   \* dynamically accessing methods on Array prototype   \*/  var arrayProto = Array.prototype;  var arrayMethods = Object.create(arrayProto);  ['push', 'pop', 'shift', 'unshift', 'splice', 'sort', 'reverse'].forEach(function (method) {   // cache original method   var original = arrayProto[method];   def(arrayMethods, method, function mutator () {   var arguments$1 = arguments;   // avoid leaking arguments:   // <http://jsperf.com/closure-with-arguments>   var i = arguments.length;   var args = new Array(i);   while (i--) {    args[i] = arguments$1[i];   }   var result = original.apply(this, args);   var ob = this.\_\_ob\_\_;   var inserted;   switch (method) {    case 'push':    inserted = args;    break    case 'unshift':    inserted = args;    break    case 'splice':    inserted = args.slice(2);    break   }   if (inserted) { ob.observeArray(inserted); }   // notify change   ob.dep.notify();   return result   });  }); |

看的出来我们模拟了一个数组对象，代理了push/pop/shift/unshift/splice/sort/reverse方法，用于检测数组的变化，并通知所有订阅者更新。如果有新建元素，会补充监听新对象。

这就是从代码上解释为什么Vue不支持数组下标修改和长度修改的原因，至于为什么这么设计，我后面会再次更新或再开篇文章，讲一些通用的设计问题以及Js机制和缺陷。

**总结**

从上面的代码中我们可以一步步由深到浅的看到Vue是如何设计出双向数据绑定的，最主要的两点：

1. 使用getter/setter代理值的读取和赋值，使得我们可以控制数据的流向。
2. 使用观察者模式设计，实现了指令和数据的依赖关系以及触发更新。
3. 对于数组，代理会修改原数组对象的方法，并触发更新。

明白了这些原理，其实你也可以实现一个简单的数据绑定，造一个小轮子，当然，Vue的强大之处不止于此，我们后面再来聊一聊它的组件和渲染，看它是怎么一步一步将我们从DOM对象的魔爪里拯救出来的。

好了，以上就是这篇文章的全部内容了，希望本文的内容对大家的学习或者工作能带来一定的帮助，如果有疑问大家可以留言交流，谢谢大家对脚本之家的支持。