# 理解vue实现原理，实现一个简单的Vue框架

参考:   
[剖析Vue实现原理 - 如何实现双向绑定mvvm](https://github.com/DMQ/mvvm)   
[Vue.js源码（1）：Hello World的背后](https://segmentfault.com/a/1190000006866881)   
[Vue.js官方工程](https://github.com/vuejs/vue)

本文所有代码可以在[git上找到](https://github.com/fwing1987/MyVue)。

也可以直接访问github pages来看看效果：<https://fwing1987.github.io/MyVue/>

       其实对JS我研究不是太深，用过很多次，但只是实现功能就算了。最近JS实在是太火，从前端到后端，应用越来越广泛，各种框架层出不穷，忍不住也想赶一下潮流。   
       Vue是近年出的一个前端构建数据驱动的web界面的库，主要的特色是响应式的数据绑定，区别于以往的命令式用法。也就是在var a=1;的过程中，拦截’=’的过程，从而实现更新数据，web视图也自动同步更新的功能。而不需要显式的使用数据更新视图（命令式）。这种用法我最早是在VC MFC中见过的，控件绑定变量，修改变量的值，输入框也同步改变。   
       Vue的官方文档，网上的解析文章都很详细，不过出于学习的目的，还是了解原理后，自己实现一下记忆深刻，同时也可以学习下Js的一些知识。搞这行的，一定要多WTFC(Write The Fucking Code)。

## 一、思考设计

       其实这里的思考是在看过几篇文章、看过一些源码后补上的，所以有的地方会有上帝视角的意思。但是这个过程是必须的，以后碰到问题就会有思考的方向。   
       先看看我们想要实现什么功能，以及现在所具有的条件：   
效果图如下：   


使用Vue框架代码如下：

<!DOCTYPE html><html lang="en"><head>

<meta charset="UTF-8">

<title>MVVM</title></head><body><script src="src/vue.js"></script><div id="msg">

{{b.c}}这是普通文本{{b.c+1+message}}这是普通文本

<p>{{message}}</p>

<p><input type="text" v-model="message"/></p>

<p>{{message}}</p>

<p><button type="button" v-on:click="clickBtn(message)">click me</button></p></div><script>

var vm = new Vue({

el:"#msg",

data:{

b:{

c:1

},

message:"hello world"

},

methods:{

clickBtn:function(message){

vm.message = "clicked";

}

}

});

</script></body></html>

然后我们还知道一个条件，Vue的官方文档所说的：

把一个普通对象传给 Vue 实例作为它的 data 选项，Vue.js 将遍历它的属性，用 Object.defineProperty 将它们转为 getter/setter。这是 ES5 特性，不能打补丁实现，这便是为什么 Vue.js 不支持 IE8 及更低版本。

用这个特性实现这样的功能，我们需要做什么呢？

1. 首先，需要利用Object.defineProperty，将要观察的对象，转化成getter/setter，以便拦截对象赋值与取值操作，称之为Observer；
2. 需要将DOM解析，提取其中的指令与占位符，并赋与不同的操作，称之为Compiler；
3. 需要将Compile的解析结果，与Observer所观察的对象连接起来，建立关系，在Observer观察到对象数据变化时，接收通知，同时更新DOM，称之为Watcher；
4. 最后，需要一个公共入口对象，接收配置，协调上述三者，称为Vue;

## 二、实现Observer

### 1.转化getter/setter

       本来以为实现起来很简单，结果只是转换为getter和setter就碰到了很多问题。原来对JS真得是只知道点皮毛啊……

开始Observer.js代码如下：

*/\*\**

*Observer是将输入的Plain Object进行处理,利用Object.defineProperty转化为getter与setter,从而在赋值与取值时进行拦截*

*这是Vue响应式框架的基础*

*\*/*function isObject(obj){

return obj != null && typeof(obj) == 'object';

}function isPlainObject(obj){

return Object.prototype.toString(obj) == '[object Object]';

}

function observer(data){

if(!isObject(data) || !isPlainObject(data)){

return;

}

return new Observer(data);

}

var Observer = function(data){

this.data = data;

this.transform(data);

};

Observer.prototype.transform = function(data){

for(var key in data){

var value = data[key];

Object.defineProperty(data,key,{

enumerable:true,

configurable:true,

get:function(){

console.log("intercept get:"+key);

return value;

},

set:function(newVal){

console.log("intercept set:"+key);

if(newVal == value){

return;

}

data[key] = newVal;

}

});

*//递归处理*

this.transform(value);

}

};

index.html:

<script src="src/Observer.js"></script><div id="msg">

<p>{{message}}</p>

<p><input type="text" v-model="message"/></p>

<p>{{message}}</p>

<p><button type="button" v-on:click="clickBtn">click me</button></p></div><script>

var a = {

b:{c:1},

d:2

};

observer(a);

a.d = 3;</script>

浏览器执行直接死循环栈溢出了，问题出在set函数里，有两个问题：

set:function(newVal){

console.log("intercept set:"+key);

if(newVal == value){

return;

}

*//这里，通过data[key]来赋值，因为我们对data对象进行了改造，set中又会调用set函数，就会递归调用，死循环*

*//而上面本来用来判断相同赋值不进行处理的逻辑，也因为value的值没有改变，没有用到。很低级的错误！*

data[key] = newVal;

}

修改为value = newVal可以吗？为什么可以这样修改，因为JS作用域链的存在，value对于这个匿名对象来说，是如同全局变量的存在，在set中修改后，在get中也可正常返回修改后的值。

但是仅仅这样是不够的，因为一个很常见的错误，在循环中建立的匿名对象，使用外部变量用的是循环最终的值！！！

还是作用域链的原因，匿名对象使用外部变量，不是保留这个变量的值，而是延长外部变量的生命周期，在该销毁时也不销毁（所以容易形成内存泄露），所以匿名对象被调用时，用的外部变量的值，是取决于变量在这个时刻的值（一般是循环执行完的最终值，因为循环结束后才有匿名函数调用）。

所以，打印a.b的值，将会是2

所以，最终通过新建函数的形式，Observer.js如下：

Observer.prototype.transform = function(data){

for(var key in data){

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

}

};

Observer.prototype.defineReactive = function(data,key,value){

var dep = new Dep();

Object.defineProperty(data,key,{

enumerable:true,

configurable:false,

get:function(){

console.log("intercept get:"+key);

if(Dep.target){

*//JS的浏览器单线程特性，保证这个全局变量在同一时间内，只会有同一个监听器使用*

dep.addSub(Dep.target);

}

return value;

},

set:function(newVal){

console.log("intercept set:"+key);

if(newVal == value){

return;

}

*//利用闭包的特性,修改value,get取值时也会变化*

*//不能使用data[key]=newVal*

*//因为在set中继续调用set赋值，引起递归调用*

value = newVal;

*//监视新值*

observer(newVal);

dep.notify(newVal);

}

});

*//递归处理*

observer(value);

};

### 2.监听队列

       现在我们已经可以拦截对象的getter/setter，也就是对象的赋值与取值时我们都会知道，知道后需要通知所有监听这个对象的Watcher，数据发生了改变，需要进行更新DOM等操作，所以我们需要维护一个监听队列，所有对该对象有兴趣的Watcher注册进来，接收通知。这一部分之前看了Vue的实现，感觉也不会有更巧妙的实现方式了，所以直接说一下实现原理。

1. 首先，我们拦截了getter；
2. 我们要为a.d添加Wacher监听者tmpWatcher；
3. 将一个全局变量赋值target=tmpWatcher；
4. 取值a.d，也就调用到了a.d的getter；
5. 在a.d的getter中，将target添加到监听队列中；
6. target = null;

       就是这么简单，至于为什么可以这样做，**是因为JS在浏览器中是单线程执行的！！所以在执行这个监听器的添加过程时，决不会有其他的监听器去修改全局变量target！！**所以这也算是因地制宜吗0\_0

       详细代码可以去看github中源码的实现，在Observer.js中。当然他还有比较复杂的依赖、剔重等逻辑，我这里只是简单实现一个。

var Dep = function(){

this.subs = {};

};

Dep.prototype.addSub = function(target){

if(!this.subs[target.uid]) {

*//防止重复添加*

this.subs[target.uid] = target;

}

};

Dep.prototype.notify = function(newVal){

for(var uid in this.subs){

this.subs[uid].update(newVal);

}

};

Dep.target = null;

## 

## 三.实现Compiler

       这里，是在看过DMQ的源码后，自己实现的一份代码，因为对JS不太熟悉，犯了一些小错误。果然学习语言的最好方式就是去写~\_~，之后，对JS的理解又加深了不少。   
       又因为想要实现的深入一点，也就是不只是单纯的变量占位符如{{a}}，而是表达式如{{a+Math.PI+b+fn(a)}}，想不出太好的办法，又去翻阅了Vue的源码实现，发现Vue的实现其实也不怎么优雅，但确实也没有更好的办法。有时候，不得不写出这种代码，如枚举所有分支，是最简单、最直接，也往往是最好的方法。

### 1.最简单的实现

       也就是纯的变量占位，这个大家都想得到，用正则分析占位符，将这个变量添加监听，与前面建立的setter/getter建立关系即可。

### 2.进阶的实现——Vue

说一下Vue的实现方法：

#### 原理：

* 将表达式{{a+Math.PI+b+fn(a)}}，变成函数：

function getter(scope) {

return scope.a + Math.PI + scope.b + scope.fn(scope.a);

}

* 调用时，传入Vue对象getter(vm)，这样，所有表达式中的变量、函数，变成vm作用域内的调用。

#### Vue的实现

var body = exp.replace(saveRE, save).replace(wsRE, '');   
\* 利用了几个正则，首先将所有的字符串提取出来，进行替换，因为后面要去除所有的空格；   
\* 去除空格；   
body = (' ' + body).replace(identRE, rewrite).replace(restoreRE, restore);   
\* 将所有的变量前加scope（除了保留字如Math,Date,isNaN等，具体见代码中的正则）；   
\* 将所有字符串替换回去   
\* 生成上面提到过的函数

可以看出这个操作还是稍微有点耗时，所以Vue做了一些优化，加了一个缓存。

### 3.实现中碰到的问题

* 明白了一个概念，DOM中每一个文字块，也是一个节点：文字节点，而且只要被其他节点分隔，就是不同的文字节点；
* JS中，可以使用childNodes与attributes等来枚举子节点与属性列表等；
* [].forEach.call，可以用来遍历非Array对象如childNodes；
* [].slice会生成数组的一个浅复制，因为childNodes在修改DOM对象时，会实时变动，所以不能直接在遍历中修改DOM，此时，可以生成浅复制数组，用来遍历；

具体代码太长就不展示，可以直接看git上的源码。

## 四、实现Watcher

       Watcher的实现，需要考虑几个问题：

* 传入的表达式如前面提到的{{a+Math.PI+b+fn(a)}}，如何与每一个具体对象建立关系，添加监听；
* 添加后的关系如何维护，其中包括：
  + 上一层对象被直接赋值，如表达式是{{a.b.c}},进行赋值a.b={c:4}，此时，c的getter没有被触发，与c相关的Watcher如何被通知；
  + 还是上面的例子，新添加的c如何与老的c的Watcher建立关系；

       其实，上面说监听队列时，已经稍微提过，利用JS单线程的特性，在调用对象的getter前，将Dep.target这个全局变量修改为Watcher，然后getter中将其添加到监听队列中。所以，Watcher中，只需要取一次表达式的值，就会实现这个功能，而且，Watcher在初始化时，本来就需要调用一次取值来初始化DOM！

       来看一下上面的问题：

* 首先，Watcher需要监听的是一个表达式，所有表达式中的成员，都需要监听，如{{a+Math.PI+b+fn(a)}}需要监听a和b的变化，而取这个表达式值时，会调用a和b的getter，从而将自身添加到a和b的监听队列中！
* 关于添加后关系的维护：
  + 我们在取表达式值{{a.b.c}}时，a和b和c的getter都会被调用，从而都会将Watcher添加到自己的监听队列中，所以a.b={c:4}赋值时，Watcher同样会被触发！
  + 上面Watcher被触发后，会重新获取a.b.c的值，则新的c的getter会被调用，从而新的c会将Watcher添加到自己的监听队列中。

       可以发现，上面的问题都被圆满解决，如果这是我自己想出来的方案，我会被自己感动哭的T\_T 这才是优雅的解决方案！

## 五、实现Vue

       这就是一个公共入口，整个框架从这里创建。需要实现的目标：

* 进行流程的串接，observe对象，compile Dom；
* 对自己的对象data，函数methods等进行代理，从而可以直接使用vm.a，vm.init等进行调用，同样通过Object.defineProperty进行对象定义；

       具体实现比较简单，可以直接参考源码。

### 六、总结

       在VUE和DMQ的基础上，实现了自己的Vue简单实现，中间碰到了很多问题，加深了对JS语言的了解，也稍微接触了流行前端框架Vue的架构实现，有兴趣的可以多看一下[源码](https://github.com/fwing1987/MyVue)。