> 相信只要你去面试vue，都会被问到vue的双向数据绑定，你要是就说个mvvm就是视图模型模型视图，只要数据改变视图也会同时更新！那你离被pass就不远了！

- 几种实现双向绑定的做法
- MVVM原理
- 实现指令解析器Compile
  - 初始化
  - 优化编译使用文档碎片
  - 编译模板
  - 编译元素
  - 编译文本
  - 处理元素/处理文本/处理事件....
- 实现一个数据监听器Observer
- 实现一个Watcher
- 代理proxy
- 面试题

几种实现双向绑定的做法

目前几种主流的mvc(vm)框架都实现了单向数据绑定，而我所理解的双向数据绑定无非就是在单向绑定的基础上给可输入元素（input、textare等）添加了change(input)事件，来动态修改model和 view，并没有多高深。所以无需太过介怀是实现的单向或双向绑定。

实现数据绑定的做法有大致如下几种：

> 发布者-订阅者模式（backbone.js）
>
> 脏值检查（angular.js）

> 数据劫持（vue.js）

发布者-订阅者模式： 一般通过sub，pub的方式实现数据和视图的绑定监听，更新数据方式通常做法是 `vm.set('property', value)`，这里有篇文章讲的比较详细，有兴趣可点这里

这种方式现在毕竟太low了，我们更希望通过 `vm.property = value` 这种方式更新数据，同时自动更新视图，于是有了下面两种方式

脏值检查： angular.js 是通过脏值检测的方式比对数据是否有变更，来决定是否更新视图，最简单的方式就是通过 `setInterval()` 定时轮询检测数据变动，当然Google不会这么low，angular只有在指定的事件触发时进入脏值检测，大致如下：

- DOM事件，譬如用户输入文本，点击按钮等。(ng-click)
- XHR响应事件 ($http)
- 浏览器Location变更事件 ($location)
- Timer事件($timeout , $interval)
- 执行 $digest() 或 $apply()

数据劫持： vue.js 则是采用数据劫持结合发布者-订阅者模式的方式，通过 `Object.defineProperty()` 来劫持各个属性的 `setter`， `getter`，在数据变动时发布消息给订阅者，触发相应的监听回调。

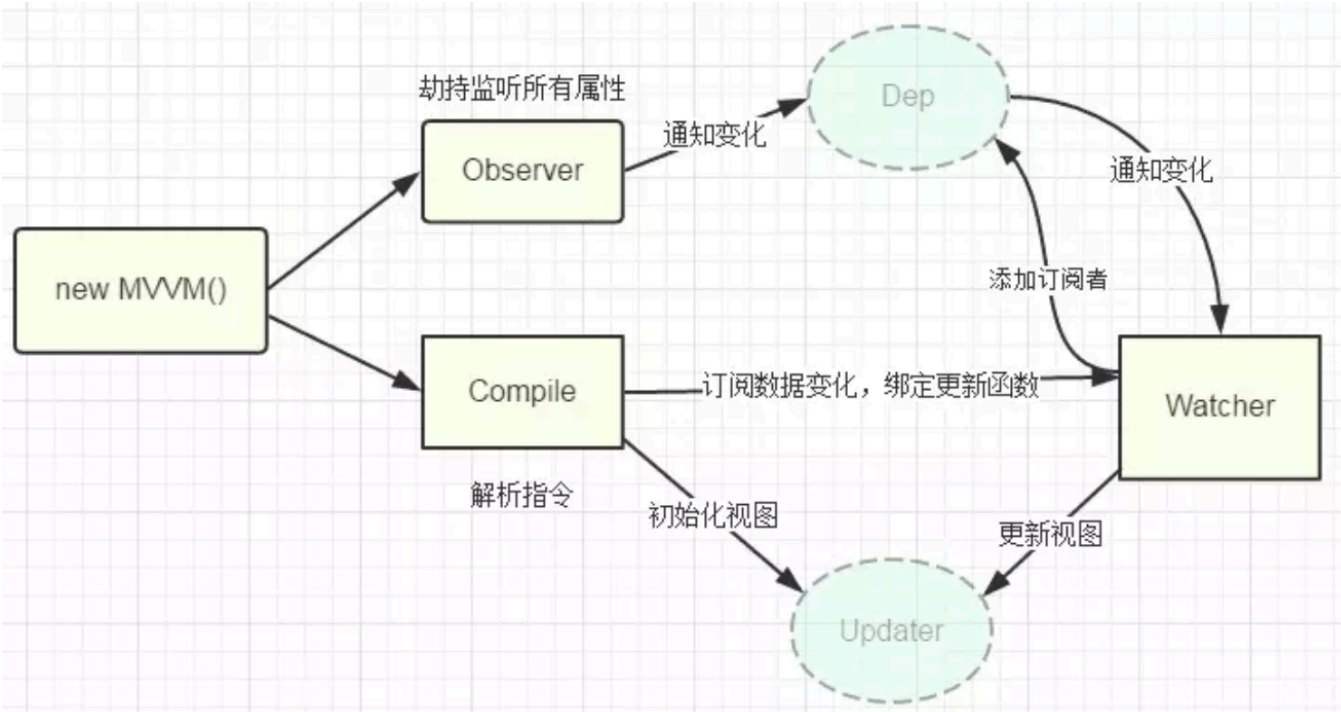
MVVM原理

Vue响应式原理最核心的方法便是通过Object.defineProperty()来实现对属性的劫持，达到监听数据变动的目的，无疑这个方法是本文中最重要、最基础的内容之一

整理了一下，要实现mvvm的双向绑定，就必须要实现以下几点：

- 1、实现一个数据监听器Observer，能够对数据对象的所有属性进行监听，如有变动可拿到最新值并通知订阅者

- 2、实现一个指令解析器Compile，对每个元素节点的指令进行扫描和解析，根据指令模板替换数据，以及绑定相应的更新函数
- 3、实现一个Watcher，作为连接Observer和Compile的桥梁，能够订阅并收到每个属性变动的通知，执行指令绑定的相应回调函数，从而更新视图
- 4、mvvm入口函数，整合以上三者



先看之前vue的功能

```
<!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>Document</title>
</head>

<body>
    <div id="app">
        <h2>{{obj.name}}--{{obj.age}}</h2>
        <h2>{{obj.age}}</h2>
        <h3 v-text='obj.name'></h3>
        <h4 v-text='msg'></h4>
        <ul>
            <li>1</li>
            <li>2</li>
            <li>3</li>
        </ul>
        <h3>{{msg}}</h3>
        <div v-html='htmlStr'></div>
        <div v-html='obj.fav'></div>
        <input type="text" v-model='msg'>
        <img v-bind:src="imgSrc" v-bind:alt="altTitle">
        <button v-on:click='handlerClick'>按钮1</button>
        <button v-on:click='handlerClick2'>按钮2</button>
```

```
        <button @click='handlerClick2'>按钮3</button>
    </div>
    <script src="./vue.js"></script>
    <script>
        let vm = new MVue({
            el: '#app',
            data: {
                obj: {
                    name: '小马哥',
                    age: 19,
                    fav:'<h4>前端Vue</h4>'
                },
                msg: 'MVVM实现原理',
                htmlStr:"<h3>hello MVVM</h3>",

 imgSrc:'https://timgsa.baidu.com/timg?image&quality=80&size=b9999_10000&sec=1568782284688&di=8635d17d550631caabfeb4306b5d76fa&imgtype=0&src=http%3A%2F%2Fh.hiphotos.baidu.com%2Fimage%2Fpic%2Fitem%2Fb3b7d0a20cf431ad7427dfad4136acaf2fdd98a9.jpg',
                altTitle:'眼睛',
                isActive:'true'

            },
```

```
48              methods: {
49                  handlerClick() {
50                      alert(1);
51                      console.log(this);
52
53                  },
54                  handlerClick2(){
55                      console.log(this);
56                      alert(2)
57                  }
58              }
59
60          })
61      </script>
62  </body>
63
64  </html>
```

实现指令解析器Compile

实现一个指令解析器Compile，对每个元素节点的指令进行扫描和解析，根据指令模板替换数据，以及绑定相应的更新函数,添加监听数据的订阅者，一旦数据有变动，收到通知，更新视图，如图所示：

初始化

新建MVue.js

```
class MVue {
    constructor(options) {
        this.$el = options.el;
        this.$data = options.data;
        //保存 options参数,后面处理数据要用到
        this.$options = options;
        // 如果这个根元素存在则开始编译模板
        if (this.$el) {
            // 1.实现一个指令解析器compile
            new Compile(this.$el, this)
        }
    }
}
class Compile{
    constructor(el,vm) {
        // 判断el参数是否是一个元素节点,如果是直接赋值,如果不是 则获取赋值
        this.el = this.isElementNode(el) ? el : document.querySelector(el);
        this.vm = vm;
    }
    isElementNode(node){
        // 判断是否是元素节点
```

```
        return node.nodeType === 1
    }
}
```

这样外界可以这样操作

```
let vm = new Vue({
    el:'#app'
})
//or
let vm = new Vue({
    el:document.getElementById('app')
})
```

优化编译使用文档碎片

```
<h2>{{obj.name}}--{{obj.age}}</h2>
<h2>{{obj.age}}</h2>
<h3 v-text='obj.name'></h3>
<h4 v-text='msg'></h4>
<ul>
    <li>1</li>
    <li>2</li>
    <li>3</li>
</ul>
<h3>{{msg}}</h3>
```

```
<div v-html='htmlStr'></div>
<div v-html='obj.fav'></div>
<input type="text" v-model='msg'>
<img v-bind:src="imgSrc" v-bind:alt="altTitle">
<button v-on:click='handlerClick'>按钮1</button>
<button v-on:click='handlerClick2'>按钮2</button>
<button @click='handlerClick2'>按钮3</button>
```

> 接下来,找到子元素的值,比如obj.name,obj.age,obj.fav 找到obj 再找到fav,获取数据中的值替换掉
>
> 但是在这里我们不得不想到一个问题,每次找到一个数据替换,都要重新渲染一遍,可能会造成页面的回流和重绘,那么我们最好的办法就是把以上的元素放在内存中,在内存中操作完成之后,再替换掉.

```
class Compile {
    constructor(el, vm) {
        // 判断el参数是否是一个元素节点,如果是直接赋值,如果不是 则获取赋值
        this.el = this.isElementNode(el) ? el : document.querySelector(el);
        this.vm = vm;
        // 因为每次匹配到进行替换时,会导致页面的回流和重绘,影响页面的性能
```

```
7           // 所以需要创建文档碎片来进行缓存,减少页面的回流和重绘
8           // 1.获取文档碎片对象
9           const fragment = this.node2Fragment(this.el);
10          // console.log(fragment);
11          // 2.编译模板
12          // 3.把子元素的所有内容添加到根元素中
13          this.el.appendChild(fragment);
14
15      }
16      node2Fragment(el) {
17          const fragment = document.createDocumentFragment();
18          let firstChild;
19          while (firstChild = el.firstChild) {
20              fragment.appendChild(firstChild);
21          }
22          return fragment
23      }
24      isElementNode(el) {
25          return el.nodeType === 1;
26      }
27  }
```

这时候会发现页面跟之前没有任何变化,但是经过Fragment的处理,优化页面渲染性能

编译模板

```
// 编译数据的类
class Compile {
    constructor(el, vm) {
        // 判断el参数是否是一个元素节点,如果是直接赋值,如果不是 则获取赋值
        this.el = this.isElementNode(el) ? el : document.querySelector(el);
        this.vm = vm;
        // 因为每次匹配到进行替换时,会导致页面的回流和重绘,影响页面的性能
        // 所以需要创建文档碎片来进行缓存,减少页面的回流和重绘
        // 1.获取文档碎片对象
        const fragment = this.node2Fragment(this.el);
        // console.log(fragment);
        // 2.编译模板
        this.compile(fragment)

        // 3.把子元素的所有内容添加到根元素中
        this.el.appendChild(fragment);

    }
```

```
    compile(fragment) {
        // 1.获取子节点
        const childNodes = fragment.childNodes;
        // 2.遍历子节点
        [...childNodes].forEach(child => {

            // 3.对子节点的类型进行不同的处理
            if (this.isElementNode(child)) {
                // 是元素节点
                // 编译元素节点
                // console.log('我是元素节点',child);
                this.compileElement(child);
            } else {
                // console.log('我是文本节点',child);
                this.compileText(child);
                // 剩下的就是文本节点
                // 编译文本节点
            }
            // 4.一定要记得,递归遍历子元素
            if (child.childNodes && child.childNodes.length) {
                this.compile(child);
            }
        })

```

```
42      }
43      // 编译文本的方法
44      compileText(node) {
45          console.log('编译文本')
46
47      }
48      node2Fragment(el) {
49          const fragment =
    document.createDocumentFragment();
50          // console.log(el.firstChild);
51          let firstChild;
52          while (firstChild = el.firstChild) {
53              fragment.appendChild(firstChild);
54          }
55          return fragment
56      }
57      isElementNode(el) {
58          return el.nodeType === 1;
59      }
60  }
```

接下来根据不同子元素的类型进行渲染

编译元素

```
compileElement(node) {
    // 获取该节点的所有属性
    const attributes = node.attributes;
    // 对属性进行遍历
    [...attributes].forEach(attr => {
        const { name, value } = attr; //v-text v-model   v-on:click  @click
        // 看当前name是否是一个指令
        if (this.isDirective(name)) {
            //对v-text进行操作
            const [, directive] = name.split('-'); //text model html
            // v-bind:src
            const [dirName, eventName] = directive.split(':'); //对v-on:click 进行处理
            // 更新数据
            compileUtil[dirName] && compileUtil[dirName](node, value, this.vm, eventName);
            // 移除当前元素中的属性
            node.removeAttribute('v-' + directive);

        }else if(this.isEventName(name)){
```

```
            // 对事件进行处理 在这里处理的是@click
            let [,eventName] =
 name.split('@');
            compileUtil['on'](node, value,
this.vm, eventName)
        }

    })

}
// 是否是@click这样事件名字
isEventName(attrName){
    return attrName.startsWith('@')
}
//判断是否是一个指令
isDirective(attrName) {
    return attrName.startsWith('v-')
}
```

编译文本

```
// 编译文本的方法
compileText(node) {
    const content = node.textContent;
    // 匹配{{xxx}}的内容
    if (/\{\{(.+?)\}\}/.test(content)) {
        // 处理文本节点
        compileUtil['text'](node, content, this.vm)
    }

}
```

大家也会发现，`compileUtil` 这个对象它是什么鬼?真正的编译操作我将它放入到这个对象中,根据不同的指令来做不同的处理.比如v-text是处理文本的 v-html是处理html元素 v-model是处理表单数据的.....

这样我们在当前对象 `compileUtil` 中通过updater函数来初始化视图

处理元素/处理文本/处理事件....

```
const compileUtil = {
    // 获取值的方法
    getVal(expr, vm) {


```

```
4           return expr.split('.').reduce((data, currentVal) => {
5               return data[currentVal]
6           }, vm.$data)
7       },
8       getAttrs(expr,vm){
9
10      },
11      text(node, expr, vm) { //expr 可能是{{obj.name}}--{{obj.age}}
12          let val;
13          if (expr.indexOf('{{') !== -1) {
14              //
15              val = expr.replace(/\{\{(.+?)\}\}/g, (...args) => {
16                  return this.getVal(args[1], vm);
17              })
18          }else{ //也可能是v-text='obj.name' v-text='msg'
19              val = this.getVal(expr,vm);
20          }
21          this.updater.textUpdater(node, val);
22      },
23      html(node, expr, vm) {
```

```
24          // html处理 非常简单 直接取值 然后调用更新函数即可
25          let val = this.getVal(expr,vm);
26          this.updater.htmlUpdater(node,val);
27      },
28      model(node, expr, vm) {
29          const val = this.getVal(expr,vm);
30          this.updater.modelUpdater(node,val);
31      },
32      // 对事件进行处理
33      on(node, expr, vm, eventName) {
34          // 获取事件函数
35          let fn = vm.$options.methods && vm.$options.methods[expr];
36          // 添加事件 因为我们使用vue时 都不需要关心this的指向问题,这是因为源码的内部帮咱们处理了this的指向
37
 node.addEventListener(eventName,fn.bind(vm),false);
38      },
39      // 绑定属性 简单的属性 已经处理 类名样式的绑定有点复杂 因为对应的值可能是对象 也可能是数组 大家根据个人能力尝试写一下
40      bind(node,expr,vm,attrName){
41          let attrVal = this.getVal(expr,vm);
```

```
42
    this.updater.attrUpdater(node,attrName,attrVal
);
43      },
44      updater: {
45          attrUpdater(node, attrName, attrVal){
46
    node.setAttribute(attrName,attrVal);
47          },
48          modelUpdater(node,value){
49              node.value = value;
50          },
51          textUpdater(node, value) {
52
53              node.textContent = value;
54
55          },
56          htmlUpdater(node,value){
57              node.innerHTML = value;
58          }
59      }
60
61  }
```

> 通过以上操作:我们实现了一个编译器compile,用它来解析指令,通过updater初始化视图

实现一个数据监听器Observer

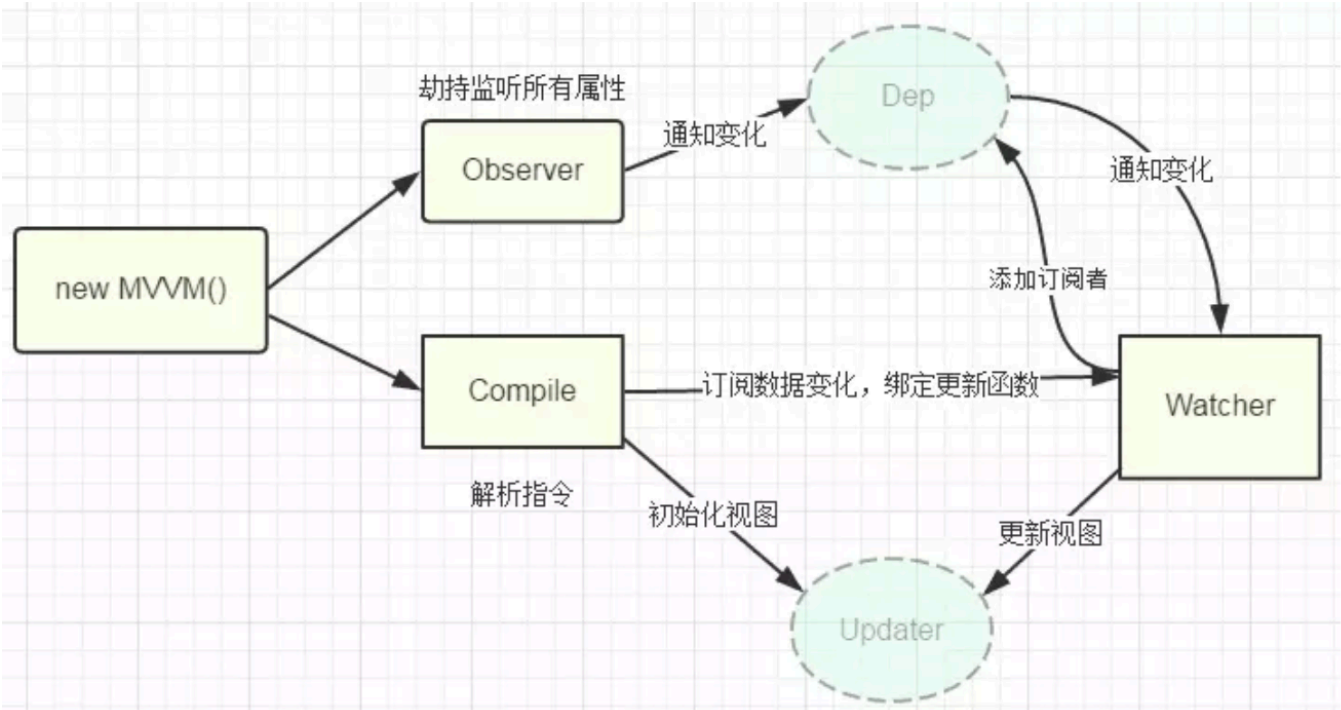
ok，思路已经整理完毕，也已经比较明确相关逻辑和模块功能了，let's do it 我们知道可以利用`Obeject.defineProperty()`来监听属性变动 那么将需要observe的数据对象进行递归遍历，包括子属性对象的属性，都加上 `setter`和`getter` 这样的话，给这个对象的某个值赋值，就会触发`setter`，那么就能监听到了数据变化。。相关代码可以是这样：

```
//test.js
let data = {name: 'kindeng'};
observe(data);
data.name = 'dmq'; // 哈哈哈，监听到值变化了 kindeng --> dmq

function observe(data) {
    if (!data || typeof data !== 'object') {
        return;
    }
    // 取出所有属性遍历
    Object.keys(data).forEach(function(key) {
        defineReactive(data, key, data[key]);
    });
};

```

```
function defineReactive(data, key, val) {
    observe(val); // 监听子属性
    Object.defineProperty(data, key, {
        enumerable: true, // 可枚举
        configurable: false, // 不能再define
        get: function() {
            return val;
        },
        set: function(newVal) {
            console.log('哈哈哈, 监听到值变化了 ', val, ' --> ', newVal);
            val = newVal;
        }
    });
}
```



再看这张图,我们接下来实现的是一个数据监听器Observer,能够对数据对象的所有属性进行监听，如有变动可拿到最新值通知依赖收集对象(Dep)并通知订阅者(Watcher)来更新视图

```
// 创建一个数据监听者  劫持并监听所有数据的变化
class Observer{
    constructor(data) {
        this.observe(data);
    }
    observe(data){
        // 如果当前data是一个对象才劫持并监听
        if(data && typeof data === 'object'){
            // 遍历对象的属性做监听
            Object.keys(data).forEach(key=>{

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

```
13
14          }
15      }
16      defineReactive(obj,key,value){
17          // 循环递归 对所有层的数据进行观察
18          this.observe(value);//这样obj也能被观察了
19          Object.defineProperty(obj,key,{
20              get(){
21                  return value;
22              },
23              set:(newVal)=>{
24                  if (newVal !== value){
25                      // 如果外界直接修改对象 则对新修改的值重新观察
26                      this.observe(newVal);
27                      value = newVal;
28                      // 通知变化
29                      dep.notify();
30                  }
31              }
32          })
33      }
34  }
```

这样我们已经可以监听每个数据的变化了，那么监听到变化之后就是怎么通知订阅者了，所以接下来我们需要实现一个消息订阅器，很简单，维护一个数组，用来收集订阅者，数据变动触发notify，再调用订阅者的update方法，代码改善之后是这样：

创建Dep

- 添加订阅者
- 定义通知的方法

```
class Dep{
    constructor() {
        this.subs = []
    }
    // 添加订阅者
    addSub(watcher){
        this.subs.push(watcher);

    }
    // 通知变化
    notify(){
        // 观察者中有个update方法 来更新视图
        this.subs.forEach(w=>w.update());
    }
}
```

虽然我们已经创建了Observer,Dep(订阅器),那么问题来了，谁是订阅者？怎么往订阅器添加订阅者?

没错，上面的思路整理中我们已经明确订阅者应该是Watcher，而且`const dep = new Dep();`是在`defineReactive`方法内部定义的，所以想通过`dep`添加订阅者，就必须要在闭包内操作，所以我们可以在`getOldVal`里面动手脚：

实现一个Watcher

它作为连接Observer和Compile的桥梁，能够订阅并收到每个属性变动的通知，执行指令绑定的相应回调函数，从而更新视图

只要所做事情：

1、在自身实例化时往属性订阅器(dep)里面添加自己

2、自身必须有一个update()方法

3、待属性变动dep.notify()通知时，能调用自身的update()方法，并触发Compile中绑定的回调，则功成身退。

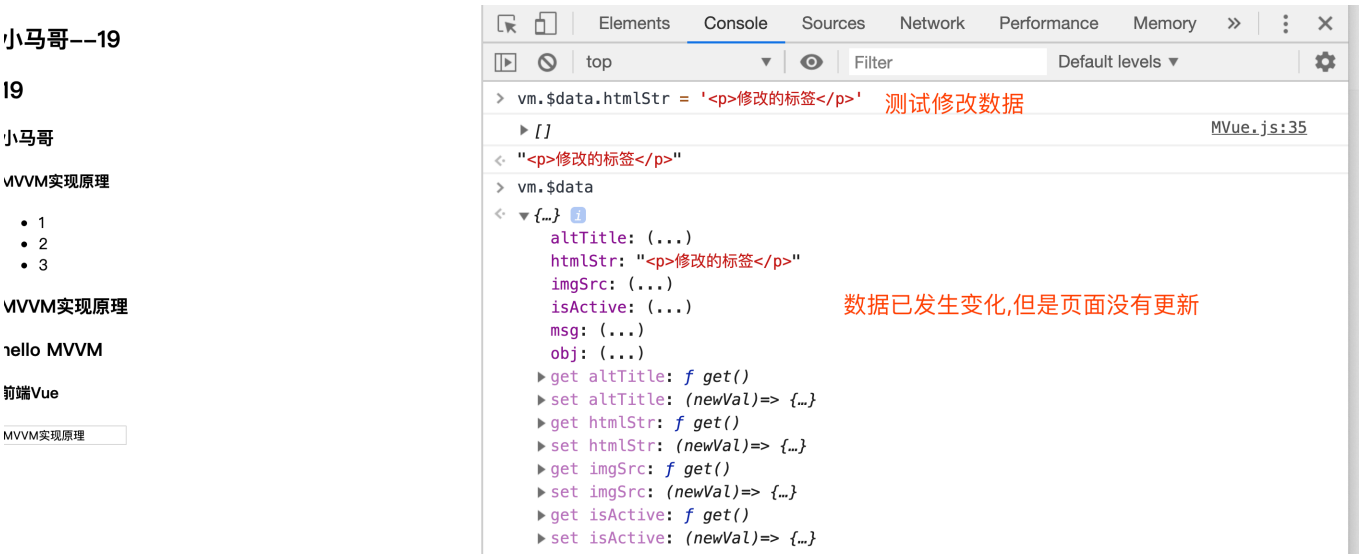
```
//Watcher.js
class Watcher{
    constructor(vm,expr,cb) {
        // 观察新值和旧值的变化,如果有变化 更新视图
        this.vm = vm;
        this.expr = expr;
        this.cb = cb;
        // 先把旧值存起来
        this.oldVal = this.getOldVal();
    }
    getOldVal(){
        Dep.target = this;

```

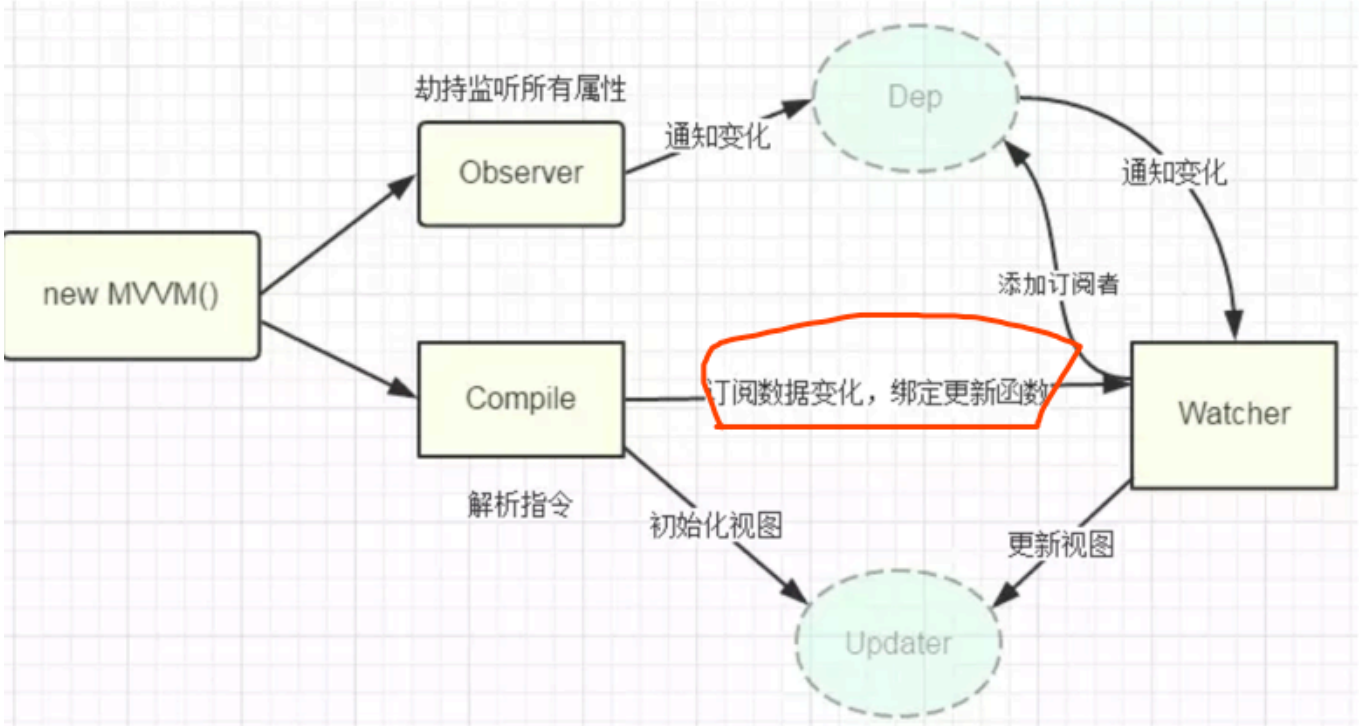
```
13          let oldVal = compileUtil.getVal(this.expr,this.vm);
14          Dep.target = null;
15          return oldVal;
16      }
17      update(){
18          // 更新操作 数据变化后 Dep会发生通知 告诉观察者更新视图
19          let newVal = compileUtil.getVal(this.expr, this.vm);
20          if(newVal !== this.oldVal){
21              this.cb(newVal);
22          }
23      }
24  }
25
26  //Observer.js
27  defineReactive(obj,key,value){
28      // 循环递归 对所有层的数据进行观察
29      this.observe(value);//这样obj也能被观察了
30      const dep = new Dep();
31      Object.defineProperty(obj,key,{
32          get(){
33              //订阅数据变化,往Dep中添加观察者
34              Dep.target && dep.addSub(Dep.target);
```

```
35                 return value;
36             },
37             //....省略
38         })
39     }
```

当我们修改某个数据时,数据已经发生了变化,但是视图没有更新



我们在什么时候来添加绑定watcher呢,继续看图



也就是说,当我们订阅数据变化时,来绑定更新函数,从而让watcher去更新视图修改

```
// 编译模板工具类
const compileUtil = {
    // 获取值的方法
    getVal(expr, vm) {
        return expr.split('.').reduce((data, currentVal) => {
            return data[currentVal]
        }, vm.$data)
    },
    //设置值
    setVal(vm,expr,val){
        return expr.split('.').reduce((data, currentVal, index, arr) => {
```

```
12              return data[currentVal] = val
13          }, vm.$data)
14      },
15      //获取新值 对{{a}}--{{b}} 这种格式进行处理
16      getContentVal(expr, vm) {
17          return expr.replace(/\{\{(.+?)\}\}/g,
(...args) => {
18              return this.getVal(args[1], vm);
19          })
20      },
21      text(node, expr, vm) { //expr 可能是
{{obj.name}}--{{obj.age}}
22          let val;
23          if (expr.indexOf('{{') !== -1) {
24              //
25              val = expr.replace(/\{\
{(.+?)\}\}/g, (...args) => {
26                  //绑定watcher从而更新视图
27                  new Watcher(vm,args[1],()=>{


28
 this.updater.textUpdater(node,this.getContentV
al(expr, vm));
29                  })
30                  return this.getVal(args[1],
vm);
```

```
            })
        }else{ //也可能是v-text='obj.name' v-text='msg'
            val = this.getVal(expr,vm);
        }
        this.updater.textUpdater(node, val);

    },
    html(node, expr, vm) {
        // html处理 非常简单 直接取值 然后调用更新函数即可
        let val = this.getVal(expr,vm);
        // 订阅数据变化时 绑定watcher,从而更新函数
        new Watcher(vm,expr,(newVal)=>{
            this.updater.htmlUpdater(node, newVal);
        })
        this.updater.htmlUpdater(node,val);
    },
    model(node, expr, vm) {
        const val = this.getVal(expr,vm);
        // 订阅数据变化时 绑定更新函数 更新视图的变化

        // 数据==>视图
        new Watcher(vm, expr, (newVal) => {
```

```
53              this.updater.modelUpdater(node, newVal);
54          })
55          // 视图==>数据
56          node.addEventListener('input',(e)=>{
57              // 设置值
58 
 this.setVal(vm,expr,e.target.value);
59 
60          },false);
61          this.updater.modelUpdater(node,val);
62      },
63      // 对事件进行处理
64      on(node, expr, vm, eventName) {
65          // 获取事件函数
66          let fn = vm.$options.methods && vm.$options.methods[expr];
67          // 添加事件 因为我们使用vue时 都不需要关心this的指向问题,这是因为源码的内部帮咱们处理了this的指向
68 
 node.addEventListener(eventName,fn.bind(vm),false);
69      },
```

```
70      // 绑定属性 简单的属性 已经处理 类名样式的绑定 有点复杂 因为对应的值可能是对象 也可能是数组 大家根据个人能力尝试写一下
71      bind(node,expr,vm,attrName){
72          let attrVal = this.getVal(expr,vm);
73
     this.updater.attrUpdater(node,attrName,attrVal);
74      },
75      updater: {
76          attrUpdater(node, attrName, attrVal){
77
     node.setAttribute(attrName,attrVal);
78          },
79          modelUpdater(node,value){
80              node.value = value;
81          },
82          textUpdater(node, value) {
83              node.textContent = value;
84          },
85          htmlUpdater(node,value){
86              node.innerHTML = value;
87          }
88      }
89
90  }
```

代理proxy

我们在使用vue的时候,通常可以直接vm.msg来获取数据,这是因为vue源码内部做了一层代理.也就是说把数据获取操作vm上的取值操作 都代理到vm.$data上

```
class Vue {
    constructor(options) {
        this.$data = options.data;
        this.$el = options.el;
        this.$options = options
        // 如果这个根元素存在开始编译模板
        if (this.$el) {
            // 1.实现一个数据监听器Observe
            // 能够对数据对象的所有属性进行监听，如有变动可拿到最新值并通知订阅者
            // Object.definerProperty()来定义
            new Observer(this.$data);

            // 把数据获取操作 vm上的取值操作 都代理到vm.$data上
            this.proxyData(this.$data);

            // 2.实现一个指令解析器Compile
            new Compiler(this.$el, this);

        }
    }
```

```
    // 做个代理
    proxyData(data){
        for (const key in data) {
            Object.defineProperty(this,key,{
                get(){
                    return data[key];
                },
                set(newVal){
                    data[key] = newVal;
                }
            })
        }
    }
}
```

面试题

闸述一下你所理解vue的MVVM响应式原理

vue.js 则是采用数据劫持结合发布者-订阅者模式的方式，通过 `Object.defineProperty()` 来劫持各个属性的 `setter`， `getter`，在数据变动时发布消息给订阅者，触发相应的监听回调。

MVVM作为数据绑定的入口，整合Observer、Compile和Watcher三者，通过Observer来监听自己的model数据变化，通过Compile来解析编译模板指令，最终利用Watcher搭起Observer和Compile之间的通信桥梁，达到数据变化 -> 视图更新；视图交互变化(input) -> 数据model变更的双向绑定效果

再配合上面的那张图,想不入职都很难