Vue2.5开发类携程旅游网App（一);

1.相关介绍;

Vue的中文官方文档比较完善, 并且发展也很快, 像NUXT这样用于Vue后端渲染的框架, 包括Weex这样使用Vue来完成移动端开发的框架都在不断推出;

更多基础内容可以参考: ‘Vue.js 2.5;’笔记;

2.Vue基础;

例子:

```
……
    <div id="root">
      <div>{{content}}</div>
    </div>
    <script>

      var app = new Vue({
        el:"#root",
        data: {
            content:'hello world'
        }
      })

      setTimeout(function(){
        app.$data.content = 'hellow world again'
      },2000)

    </script>
……
```

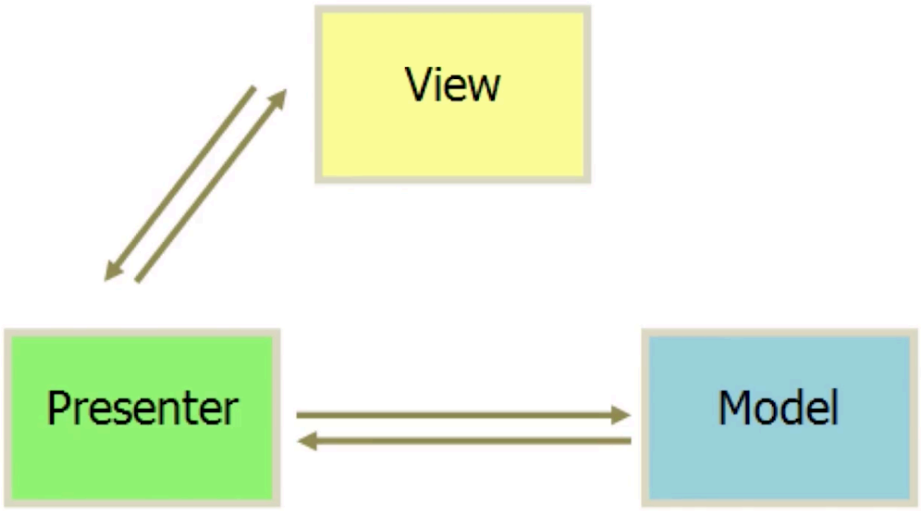
上例中在页面中加载后会显示hello world, 2秒后显示hello world again;
需要注意的是, 上例中的:app.$data.content = 'hellow world again' , 可以写成:
app.content = 'hellow world again’, 效果相同;

之前在‘Vue.js 2.5;’笔记中使用Vue完成了一个todolist的功能模块, 由于Vue是MVVM结构的框架, 而MVVM是基于MVP的框架, 这里将使用jQuery来实现MVP结构的todolist功能模块;

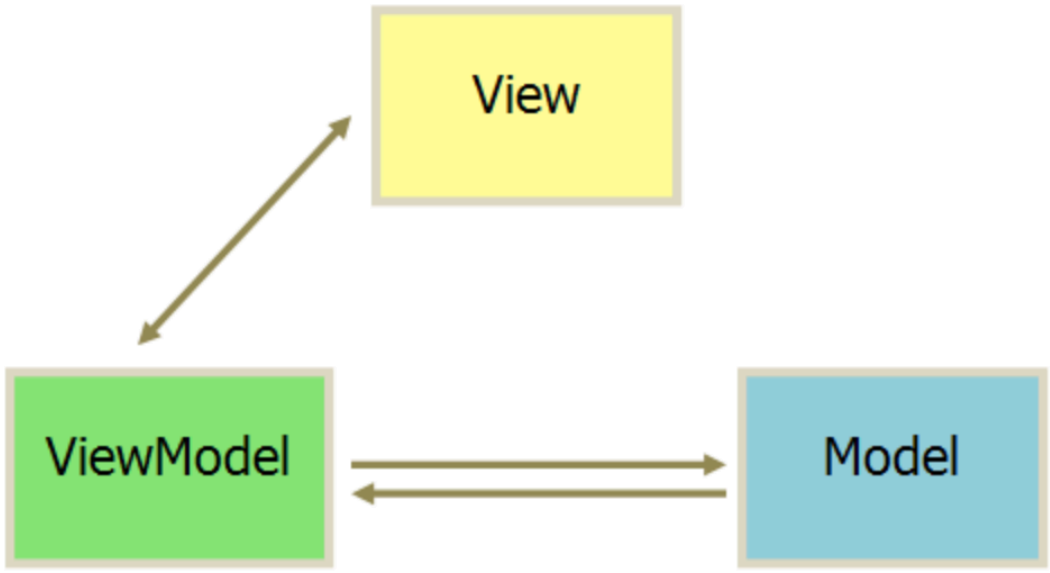
例子:

```
……
    <div id="app">
      <input id='input' />
      <button id='btn'>提交</button>
      <ul id='list'></ul>
    </div>
    <script>
        function Page(){
        }

        $.extend(Page.prototype,{
          init: function(){
            this.bindEvents()
          },
          bindEvents: function(){
            var btn = $('#btn');
            btn.on('click', $.proxy(this.handleBtnClick,this))
          },
          handleBtnClick: function(){
            var inputEle = $('#input');
            var inputValue = inputEle.val();
            var ulEle = $('#list');
            ulEle.append('<li>' + inputValue + '</li>');
            inputEle.val('');
          }
        })

        var page = new Page();
        page.init();

    </script>
……
```

上例中, 使用了jQuery来实现了基于MVP结构的todolist功能, M层并没有在上例中体现, 因为没有使用任何数据模型去获取/存储数据, V层就是html元素, P层在MVP结构项目中是核心层(其中大部分操作属于DOM操作), 它监听V层的用户交互行为并且执行相应的业务逻辑, 然后通过DOM操作去更新页面, 同时它又负责调用数据模型来完成数据的存储与获取:



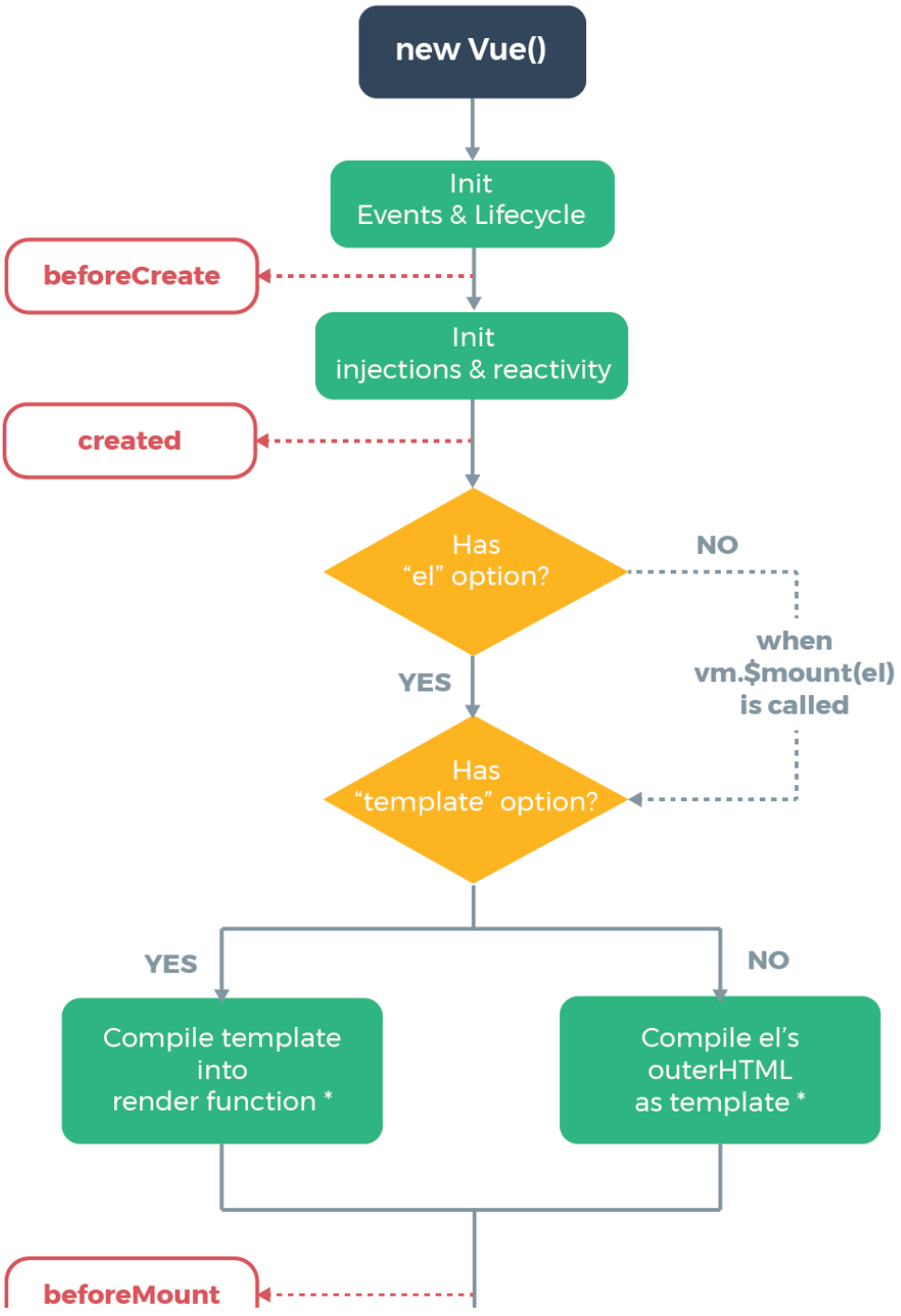
mvvm(比较典型的应用包括: .NET的WPF，js框架Knockout、AngularJS等)

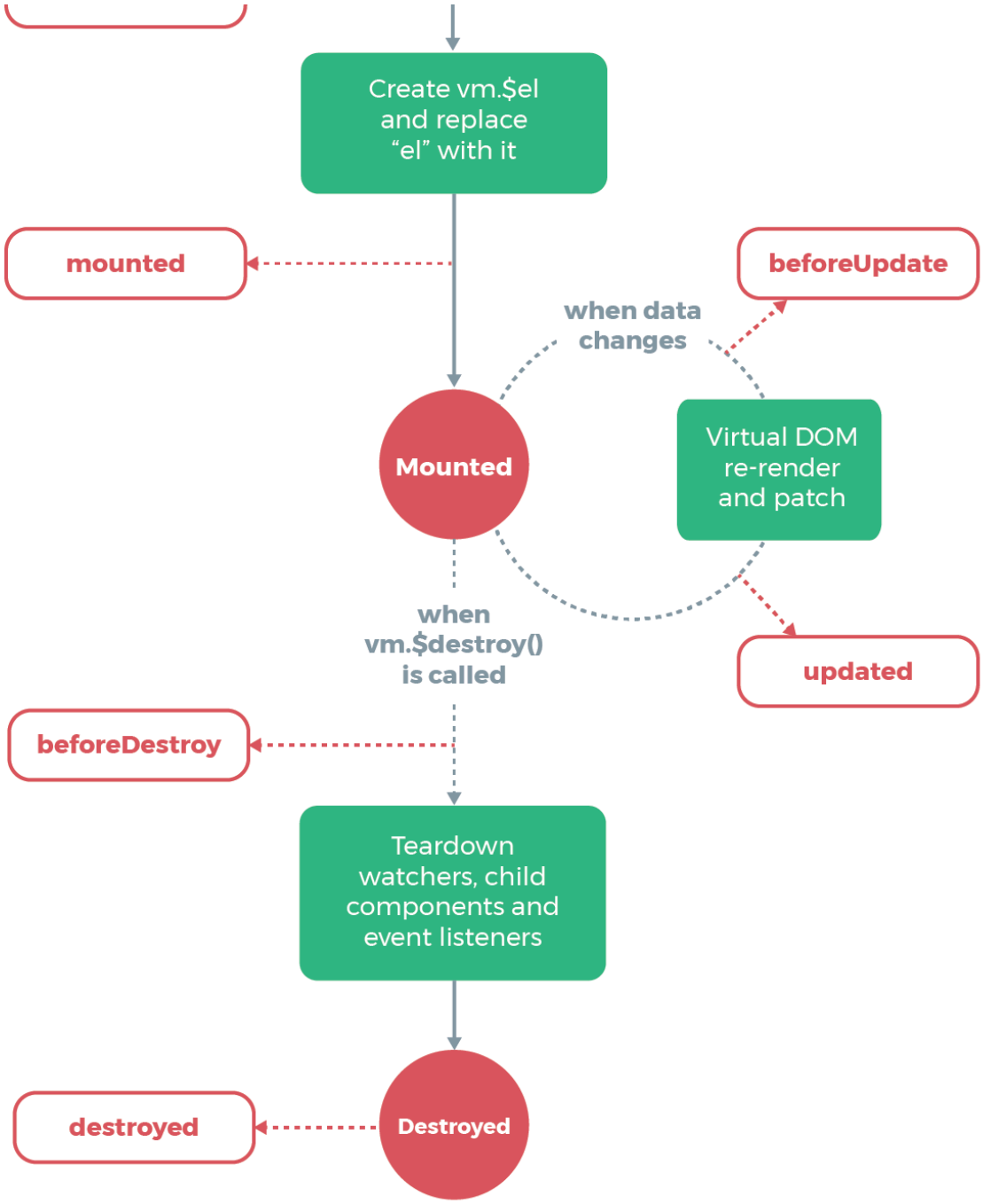


很显然, mvvm结构使用ViewModel代替了Presenter层, 其实它们的职责是类似的, 只不过使用了mvvm结构的前端框架会自动为开发者实现vm层的功能: 监听视图层的数据变

化, 并且更新到ViewModel中, 同时也会检查ViewModel中的数据变化并自动更新到视图层中, 这样就省去了原先在MVP结构项目中的大量DOM操作, 所以可以认为使用了mvvm结构框架的开发者只需要专注于对M层的开发即可(当然M层中会包含业务逻辑和对数据模型的调用);

Vue实例生命周期钩子;





* template compilation is performed ahead-of-time if using
a build step, e.g. single-file components

参考:
https://cn.vuejs.org/v2/guide/
instance.html#%E7%94%9F%E5%91%BD%E5%91%A8%E6%9C%9F%E5%9B%BE%E

7%A4%BA

例子:

```
var vm = new Vue({
        el: '#app',
        template: '<div>{{test}}</div>',
        data: {
          test: 'hello world'
        },
        beforeCreate(){
          console.log('beforeCreate')
        },
        created(){
          console.log('created')
        },
        beforeMount(){
          console.log(this.$el.innerText)  //default
          console.log('beforeMount')
        },
        mounted(){
          console.log(this.$el.innerText)  //hello world
          console.log('mounted')
          var that = this
          var des = function(){
            that.$destroy()
          }
          setTimeout(des,3000)
          this.test = 'updated'
        },
        beforeUpdate(){
          console.log('beforeUpdate')
        },
        updated(){
          console.log('updated')
        },
        beforeDestroy(){
          console.log('beforeDestroy')
        },
        destroyed(){
          console.log('destroyed')
        }
```

```
})
```

上例中在控制台输出的内容为:
beforeCreate
created
default
beforeMount
hello world
mounted
beforeUpdate
updated

延迟3秒后输出:
beforeDestroy
destroyed

需要注意的是, 当vue实例vm被destroy之后它在页面中已经显示的模板内容以及模板上已经绑定的交互事件会被保留, 但是由于实例对象本身被destroy了(这里的destroy是销毁vm实例对象上绑定的一些功能, 而这个对象本身还是存在的), 所以再次修改vm上的data属性就不会触发页面的更新了(双向数据绑定功能已被移除);

除了上面介绍的8中vue实例的生命周期钩子函数, 其实还有其它三个钩子函数:
activated, deactivated, errorCaptured
之后会介绍;

补充:
1.不要在选项属性或回调上使用**箭头函数**，比如:

```
created: () => console.log(this.a)
```

或者

```
vm.$watch('a', newValue => this.myMethod())
```

因为箭头函数是和父级上下文绑定在一起的, this 不会是如你所预期的 Vue 实例对象的引用, 经常导致 Uncaught TypeError: Cannot read property of
undefined 或 Uncaught TypeError: this.myMethod is not a function 之类的错误;

Vue的计算属性和侦听器;

例子:

```
……
    <div id="app">
      {{fullName + ' ' + age}}
```

```
    </div>
    <script>

        var vm = new Vue({
          el: '#app',
          data: {
            firstName: 'song',
            lastName: 'jiucong',
            fullName: 'song jiuchong',
            age: 29
          },
          watch: {
            firstName: function(){
              console.log('firstName改动触发侦听器')
              this.fullName = this.firstName + ' ' + this.lastName
            },
            lastName: function(){
              console.log('lastName改动触发侦听器')
              this.fullName = this.firstName + ' ' + this.lastName
            }
          },
          // computed: {
          //   fullName: function(){
          //     console.log('计算属性被计算一次')
          //     return this.firstName + ' ' + this.lastName
          //   }
          // }
        })
    </script>
……
```

上例中在页面中加载后显示: song jiuchong 29
在控制台执行: vm.age = 28 后, 页面显示更新为: song jiuchong 28
在控制台执行: vm.firstName = 'SONG' 后, 页面显示更新为: SONG jiucong 28, 并且控制台输出: firstName改动触发侦听器
在控制台执行: vm.lastName = ‘JIUCHONG’ 后, 页面显示更新为: SONG JIUCHONG 28, 并且控制台输出: lastName改动触发侦听器

如果上例中开放了vue实例的computed计算属性, 页面加载后会报错:
[Vue warn]: The computed property "fullName" is already defined in data.

所以需要相应删除vue实例的data属性中设置的fullName;

使用了计算属性后, 页面加载后显示: song jiucong 29, 控制台输出: 计算属性被计算一次
在控制台执行: vm.age = 28 后, 页面显示更新为: song jiuchong 28
在控制台执行: vm.firstName = 'SONG' 后, 页面显示更新为: SONG jiucong 28, 并且控制台输出: 计算属性被计算一次
在控制台执行: vm.lastName = 'JIUCHONG' 后, 页面显示更新为: SONG JIUCHONG 28, 并且控制台输出: 计算属性被计算一次

很显然计算属性会在this.firstName和this.lastName这两个输入值没有改变的情况下直接使用缓存的输出值, 代码更简洁, 效率更高;

计算属性的setter/getter;

例子:

```
……
    <div id="app">
      {{fullName}}
    </div>
    <script>
        var vm = new Vue({
          el: '#app',
          data: {
            firstName: 'song',
            lastName: 'jiucong'
          },
          computed: {
            fullName: {
              get: function(){
                console.log('计算属性get方法被执行')
                return this.firstName + ' ' + this.lastName
              },
              set: function(value){
                console.log('计算属性set方法被执行')
                var arr = value.split(' ')
                this.firstName = arr[0]
                this.lastName = arr[1]
              }
            }
          }
        })
    </script>
```

……

上例中在vue实例中设置了fullName这个计算属性的setter/getter方法, 所以在获取this.fullName时就会通过get方法获取返回值(同样会根据依赖进行缓存), 如果对this.fullName进行赋值操作, 就会调用set方法;
可以这样理解, 直接将某个计算属性设置为一个函数(参考之前设置计算属性的例子), 就相当于只设置了这个计算属性的get方法;
上例在页面加载后显示: song jiucong, 控制台显示: 计算属性get方法被执行
在控制台执行: vm.firstName='SONG' 后, 页面显示为: SONG jiucong, 控制台显示: 计算属性get方法被执行
在控制台执行: vm.lastName='JIUCHONG' 后, 页面显示为: SONG JIUCHONG, 控制台显示: 计算属性get方法被执行
在控制台执行: vm.fullName='jiuchong song' 后, 页面显示为: jiuchong song, 控制台显示: 计算属性set方法被执行 计算属性get方法被执行

可以发现, 当对vue实例的firstName/lastName属性进行修改时, 由于它们是计算属性fullName的依赖, 所以会重新执行一遍fullName的get方法然后更新页面内容;
当对vue实例的计算属性fullName本身进行赋值, 就会执行其set方法, 从而修改了this.firstName/this.lastName这两个它所依赖的属性, 所以又会接着触发一遍get方法, 这也是上例中执行: vm.fullName='jiuchong song' 后控制台显示: '计算属性set方法被执行' '计算属性get方法被执行' 的原因;

Vue中的样式绑定;

(1)Vue中class属性的对象绑定;

例子:

```
……
            <style>
                .activated {
                    color:red;
                }
            </style>
        </head>
        <body>
            <div id='app'>
                <div @click='handleDivClick'
                    :class='{activated: isActivated}'
                >hello world</div>
            </div>
```

```
        <script>
            var vm = new Vue({
                el:'#app',
                data: {
                    isActivated: false
                },
                methods: {
                    handleDivClick: function(){
                        this.isActivated = !this.isActivated
                    }
                }
            })
        </script>
.....
```

上例中，Vue实例通过v-bind指令绑定了元素的class属性后，使用了对象的形式为其赋值: {activated: isActivated}, 这种方式会让Vue去判断activated的值为true/false(其它类型的值将会被转换为true/false)，如果为true则在元素上设置class:activated, 如果为false则在元素上设置class(class属性没有任何内容);
所以就实现了点击div元素后其名为activated的样式会被激活/移除的效果;

当然, class属性所绑定的对象中可以有多个属性, 如果上例改为:

```
.....
:class='{activated: isActivated, activatedOne: isActivated}'
```

那么点击div元素后html中内容为:

```
<div class="activated activatedOne">hello world</div>
```

(2)Vue中class属性的数组绑定;

例子:

```
.....
        <style>
            .activated {
                color:red;
            }
        </style>
    </head>
    <body>
        <div id='app'>
            <div @click='handleDivClick'
                :class='[activated, activatedOne]'
```

```
        >hello world</div>
    </div>

    <script>
        var vm = new Vue({
            el:'#app',
            data: {
                activated: '',
                activatedOne: 'activatedOne'
            },
            methods: {
                handleDivClick: function(){
                    if(this.activated === 'activated'){
                        this.activated = ''
                    }else{
                        this.activated = "activated"
                    }
                }
            }
        })
    </script>
.....
```

上例中，Vue实例通过v-bind指令绑定了元素的class属性后，使用了数组的形式为其赋值: [activated, activatedOne], 这种情况下Vue会将数组中元素视为变量，它会将所有变量的值直接设置为元素上class属性的值，所以上例在页面加载后html中的内容为:
hello world

当点击了div元素后html中的内容为:
hello world

(3)Vue中style属性的对象绑定;

例子:

```
.....
    <div id='app'>
        <div :style="styleObj"
            @click="handleDivClick"
        >hello world</div>
    </div>

    <script>
```

```
                var vm = new Vue({
                    el:'#app',
                    data: {
                        styleObj: {
                            color: "black"
                        }
                    },
                    methods: {
                        handleDivClick: function(){
                            this.styleObj.color = this.styleObj.color === 
"black"?"red":"black"
                        }
                    }
                })
            </script>
......
```

上例中，Vue实例通过v-bind指令绑定了元素的style属性后，使用了对象的形式为其赋值: style="styleObj", 这种情况下Vue会通过style对象的形式(类似于react为组件设置style属性的方法)为这个style属性赋值, 所以上例中在页面加载后html为:
hello world

点击div元素后html为:
hello world

(4)Vue中style属性的数组绑定;

例子:
```
......
:style="[styleObj, {fontSize: '20px'}]"
......
```

上例中，Vue实例通过v-bind指令绑定了元素的style属性后，使用了数组的形式为其赋值: [styleObj, {fontSize: '20px'}], 这种情况下其实就是将多个style对象放入一个数组, 然后Vue将数组中的所有style对象的值都设置为元素中的样式;
所以上例中在页面加载后html为:
hello world

点击div元素后html为:
hello world

Vue中的条件渲染;

例子1:

```
.....
        <div id='app'>
            <div v-if="show">{{message}}</div>
            <div v-else>Bye</div>
        </div>

        <script>
            var vm = new Vue({
                el:'#app',
                data: {
                    show: false,
                    message: "Hello World"
                }
            })
        </script>
.....
```

上例中使用了v-if和v-else指令, 页面加载后显示: 'Bye'，如果将vm.show设置为true, 那么页面显示:'Hello World'；
需要注意的是, 设置了v-if和v-else指令的标签必须是连续的(当中没有其它标签), 不然会报错, 如果上例改为:

```
.....
            <div v-if="show">{{message}}</div>
            <span></span>
            <div v-else>Bye</div>
.....
```

页面中报错:

```
[Vue warn]: Error compiling template:
.....
 v-else used on element <div> without corresponding v-if.
```

例子2:

```
.....
        <div id='app'>
            <div v-if="show === 'a'">This is a</div>
            <div v-else-if="show === 'b'">This is b</div>
```

```
        <div v-else>This is others</div>
    </div>

    <script>
        var vm = new Vue({
            el:'#app',
            data: {
                show: "a",
            }
        })
    </script>
```
.....

上例中在页面显示为: This is a, 当将vm.show设置为'b'后页面显示为: 'This is b', 将vm.show设置为'c'后页面显示为: 'This is others';

例子3:

```
.....
    <div id='app'>
        <div v-if="show">
            用户名: <input />
        </div>
        <div v-else>
            邮箱名: <input />
        </div>
    </div>

    <script>
        var vm = new Vue({
            el:'#app',
            data: {
                show: false,
            }
        })
    </script>
```
.....

上例中在页面加载后会显示: '邮箱名:'和一个input输入框;
在输入框中输入一些内容后，将vm.show设置为true, 此时页面中会显示: '用户名:'和一个input输入框, 但是输入框中仍旧保留了之前输入的内容;
原因就是Vue的页面更新机制同样会利用virtual dom的diff算法来尽力少的执行dom操作, 所以对于Vue来说, 改变vm的show属性会引起的页面变化只需要将'用户名:'更新

为'邮箱名:'即可, <input/>元素在变化前后始终存在无需执行dom操作进行改变, 所以用户之前在input元素中的输入会被保留;

那么解决方法就是为<input/>元素添加key属性， 如:

```
.....
                <div v-if="show">
                    用户名: <input key="username"/>
                </div>
                <div v-else>
                    邮箱名: <input key="emailname"/>
                </div>
.....
```

这样当vm.show改变时, Vue会重新渲染页面中的<input/>元素, 那么用户之前在输入框中的输入内容就会被清空了;

Vue的页面更新机制可以参考:
'...'笔记中: ...相关内容;

Vue中的列表渲染;

(1)数组循环;

例子1:

```
.....
        <div id='app'>
            <div v-for="(item, index) of list"
                 :key="item.id">
                {{item.text}} --- {{index}}
            </div>
        </div>

        <script>
            var vm = new Vue({
                el:'#app',
                data: {
                    list:[{
                        id:"01",
                        text:"hello"
                    }, {
                        id:"02",
```

```
                    text:"world"
                }]
            }
        })
    </script>
```
.....

上例中在页面显示为:
hello --- 0
world --- 1

此时如果通过: vm.list[2] = {id:'03', text:"!"} 来改变Vue实例中list属性的内容, 那么页面不会有任何变化, 但是vm.list确实已经被添加了一项元素; 这和之前在介绍Vue的'脏检查'机制时提到的内容相关, 因为Vue在通常情况下是通过===来判断一个属性前后是否发生变化的, 但是对于一个引用类型的值而言, 只要它的地址没有变化, ===对比的结果是相同的, 所以当为一个数组添加一项新的元素时会检测不到前后的变化, 为了解决这个问题, Vue覆写了js原生数组的一些API, 使得用户在执行这些方法时能够让Vue监听到这些变化从而根据新的数据来更新页面内容;

这些被覆写的API为:
push, pop, shift, unshift, splice, sort, reverse;

需要注意的是, 与数据为对象时不同的是, 当Vue实例的一个数据为数组类型时, Vue不会去递归监听数组中每一项元素的变化, 所以如果上例在页面中加载后执行:
vm.list[0] = 'HELLO'

页面同样不会有更新, 所以Vue要覆写: splice, sort, reverse 这三个数组的API;

如果上例中在页面加载后执行: vm.list.splice(0,1,{id:'001',text:'HELLO'}), 那么页面中显示为:
HELLO --- 0
world --- 1

例子2:
```
.....
        <template v-for="(item, index) of list">
            <div>{{index}}</div>
            <span>{{item.text}}</span>
        </template>
```
.....

上例中使用了template标签来包裹两个需要被循环遍历的子元素, 它的作用是:

template标签本身不会被渲染到页面中, 所以html为:

```
<div id="app">
        <div>0</div>
        <span>hello</span>
        <div>1</div>
        <span>world</span>
</div>
```

(2)对象循环;

例子:

```
.....
            <div id='app'>
                <div v-for="(item, key, index) of userInfo">
                    {{item}} --- {{key}} --- {{index}}
                </div>
            </div>

            <script>
                var vm = new Vue({
                    el:'#app',
                    data: {
                        userInfo: {
                            name:'song',
                            age: 28,
                            gender:'male'
                        }
                    }
                })
            </script>
.....
```

上例中对vm实例中的userInfo对象进行循环遍历, 在页面中显示为:

song --- name --- 0
28 --- age --- 1
male --- gender --- 2

如果上例中改为:

```
<div v-for="(item, key, index) in userInfo">
```

效果相同; 所以Vue中淡化了for of/for in的区别;

如果在页面加载后执行:
vm.userInfo.age = 29

那么页面就会更新为:
song --- name --- 0
29 --- age --- 1
male --- gender --- 2

说明Vue实例会递归监听其对象中每一个属性的变化(与数据类型为数组时不同), 然后更新页面内容;

如果在页面加载后执行:
vm.userInfo.address = 'shanghai'

页面不会有任何变化; 说明Vue并不会监听为其数据对象添加一个新的属性这个操作的变化, 因为这种情况下数据对象本身的值没有发生变化(对象中所有已经存在并被监听的属性的值也没有发生变化);

解决方法可以是:

<1>改变这个数据对象本身的引用(或者使用immutable.js), 如:
vm.userInfo = {name:'song', age: 28, gender:'male', address:'shanghai'}

页面中显示:
song --- name --- 0
28 --- age --- 1
male --- gender --- 2
shanghai --- address --- 3

<2>使用Vue.set()方法;
上例在页面加载后执行:
Vue.set(vm.userInfo,'address','shanghai')

页面中显示同<1>;

<3>使用vue.$set()方法;
除了在<2>中介绍的使用全局的方式在Vue上调用set()方法, 在Vue实例上也可以使用set方法, 如:
上例在页面加载后执行:
vm.$set(vm.userInfo,'address','shanghai')

页面中显示同<1>;

(3)数组循环中使用Vue的set方法;

例子:

```
.....
        <div id='app'>
            <div v-for="(item) of userInfo">
                {{item}}
            </div>
        </div>

        <script>
            var vm = new Vue({
                el:'#app',
                data: {
                    userInfo: [1, 2, 3, 4]
                }
            })
        </script>
.....
```

Vue的set方法同样也可以用来修改vue实例中数组类型的数据;

上例在页面加载后执行:

```
Vue.set(vm.userInfo,'4',5)
vm.$set(vm.userInfo,'1','x')
```

页面显示为:

```
1
x
3
4
5
```

Vue组件使用的注意点;

(1)使用is属性来解决浏览器渲染html时可能出现的一些问题;

例子:

```
.....
```

```
        <div id='app'>
            <table>
                <tbody>
                    <row></row>
                    <row></row>
                    <row></row>
                </tbody>
            </table>
        </div>

        <script>
            Vue.component('row', {
                template: '<tr><td>this is a row</td></tr>'
            })
            var vm = new Vue({
                el:'#app'
            })
        </script>
.....
```

上例在页面中的html为:

```
<div id="app">
    <tr><td>this is a row</td></tr>
    <tr><td>this is a row</td></tr>
    <tr><td>this is a row</td></tr>
    <table><tbody></tbody></table>
</div>
```

可以发现浏览器对html的渲染存在问题, 这是因为h5规定在<table>的<tbody>中一定要使用相关的<tr>标签, 不然就会作为<table>同级渲染在外部, 解决办法是使用Vue的is属性, 上例修改为:

```
.....
        <div id='app'>
            <table>
                <tbody>
                    <tr is='row'></tr>
                    <tr is='row'></tr>
                    <tr is='row'></tr>
                </tbody>
            </table>
        </div>

        <script>
```

```
            Vue.component('row', {
                template: '<tr><td>this is a row</td></tr>'
            })
            var vm = new Vue({
                el:'#app'
            })
        </script>
.....
```

上例在页面中的html为:

```
<div id="app">
    <table>
        <tbody>
            <tr><td>this is a row</td></tr>
            <tr><td>this is a row</td></tr>
            <tr><td>this is a row</td></tr>
        </tbody>
    </table>
</div>
```

可以发现, 使用了is属性后, 可以让浏览器正确识别tr元素, 之后Vue会将设置了is='row'的tr元素作为row组件进行解析;

同样, 在使用
这样的元素组合时最好也为<li>元素设置is属性, 如果直接将<li>设置为组件元素, 有些版本的浏览器可能会出现渲染问题;
同理: <select><option></option></select>

(2)data属性的类型;

例子:

```
.....
    <div id='app'>
            <table>
                <tbody>
                    <tr is='row'></tr>
                    <tr is='row'></tr>
                    <tr is='row'></tr>
                </tbody>
            </table>
        </div>

        <script>
```

```
        Vue.component('row', {
            data: {
                content: 'this is a row'
            },
            template: '<tr><td>{{content}}</td></tr>'
        })
        var vm = new Vue({
            el:'#app'
        })
    </script>
.....
```

上例在页面中加载后会报错:
The "data" option should be a function that returns a per-instance value in component definitions
.....

这是因为在Vue中只有根实例可以将data属性设置为一个对象, 其它子组件都必须将data属性设置为一个返回最终数据对象的函数; 原因是: 根实例只会被初始化一次, 而其它的子组件都可能会被实例化若干次, 所以就需要在每次初始化一个组件实例时可以获取一个全新的data数据对象;

所以上例应该改为:

```
.....
data: function(){
                return {content:'this is row'}
            }
.....
```

(3)在Vue中使用ref属性来进行dom操作(机制与react中的ref属性类似);

虽然Vue已经通过双向数据绑定的机制帮我们完成了大量的dom操作, 但是在某些复杂的情况下(如: 动画演示), 还是需要我们手动操作dom元素的, 而ref属性就是帮助我们进行dom操作的;

例子1:

```
.....
    <div id='app'>
        <div ref='hello'
            @click='handlClick'>hello world</div>
    </div>
```

```
        <script>
            var vm = new Vue({
                el:'#app',
                methods: {
                    handlClick: function() {
                        console.log(this.$refs.hello.innerHTML)
                    }
                }
            })
        </script>
```
.....

上例中, 在页面中点击hello world后控制台显示:
hello world

例子2:
.....

```
        <div id='app'>
            <counter ref="counter1" @change='handleChange'></counter>
            <counter ref="counter2" @change='handleChange'></counter>
            <div>{{total}}</div>
        </div>

        <script>
            Vue.component('counter',{
                template: '<div @click="handleClick">{{number}}</div>',
                data: function(){
                    return {
                        number:0
                    }
                },
                methods: {
                    handleClick: function(){
                        this.number++
                        this.$emit('change')
                    }
                }
            })

            var vm = new Vue({
                el:'#app',
                data: {
```

```
                total: 0
            },
            methods: {
                handleChange: function(){
                    this.total = this.$refs.counter1.number + this.
$refs.counter2.number
                }
            }
        })
    </script>
.....
```

上例通过Vue的ref属性完成了一个计数器, 两个counter子组件在被点击时它们的number属性会自动加1, 而父组件会监听每一次子组件点击后的变化, 并将两个子组件的number属性的值相加后赋给自己的total属性, 从而更新页面中显示的总和;

父子组件之间的数据传递;

在 Vue 中，父子组件的关系可以总结为 prop 向下传递，事件向上传递。父组件通过 prop 给子组件下发数据，子组件通过事件给父组件发送消息;

例子1:

```
.....
<div id='app'>
        <counter count=0></counter>
        <counter :count='0'></counter>
    </div>

    <script>
        var counter = {
            props: ['count'],
            template: '<div @click="handleClick">{{count}}</div>',
            methods: {
                handleClick: function(){
                    this.count++
                }
            }
        }

        var vm = new Vue({
            el:'#app',
```

```
                components: {
                    counter: counter
                }
            })
        </script>
```
.....

上例在页面中加载后显示为:
0
0

如果点击某个子组件, 其页面中显示的值会加1; 但是需要特别注意的是, 此时控制台会报错:
[Vue warn]: Avoid mutating a prop directly since the value will be overwritten whenever the parent component re-renders. Instead, use a data or computed property based on the prop's value. Prop being mutated: "count"

这是因为Vue使用单向数据流的形式(与react类似)来更新应用中的数据, 也就是说Vue只希望从父组件传递属性给子组件, 而子组件无法将属性的变化直接反馈给父组件(防止造成数据流的混乱); 而上面的警告所表明的意思是, 如果一个子组件修改了从父组件接收到的props属性, 那么当这个props属性为引用类型时, 父组件由于数据的变化被重新render, 而如果子组件此时选择不重新render而保留当前数据, 那么子组件中的引用类型的props数据将被覆盖;
另外, 避免在子组件中修改接收到的props属性还有一个非常重要的原因, 因为如果这个props属性是引用类型的, 那么一旦某个子组件修改了这个props属性其中的内容, 那么所有接收了相同父组件传递的这个props属性的子组件都会受到影响;

上例中还有一点需要注意的是, 在counter组件上设置count=0与:count='0'的作用相同(都将传递一个数字类型而非字符串类型的值), 因为设置为:count会将'0'引号中的内容当成js表达式来解析;

为了避免报错, 将上例修改为:
```
.....
            var counter = {
                props: ['count'],
                data: function(){
                    return {
                        number: this.count
                    }
                },
                template: '<div @click="handleClick">{{number}}</div>',
                methods: {
```

```
                handleClick: function(){
                    this.number++
                }
            }
        }
```
.....

例子2:

```
.....
        <div id='app'>
            <counter count=0 @inc='handleInc'></counter>
            <counter :count='0' @inc='handleInc'></counter>
            <div>{{total}}</div>
        </div>

        <script>
            var counter = {
                props: ['count'],
                data: function(){
                    return {
                        number: this.count
                    }
                },
                template: '<div @click="handleClick">{{number}}</div>',
                methods: {
                    handleClick: function(){
                        this.number++
                        this.$emit('inc', 1)
                    }
                }
            }

            var vm = new Vue({
                el:'#app',
                components: {
                    counter: counter
                },
                data: {
                    total:0
                },
                methods: {
                    handleInc: function(step){
```

```
                        this.total += step
                    }
                }
            })
        </script>
```
.....

组件参数校验与非props特性;

例子1:
```
.....
    <div id='app'>
            <child content='hello world'></child>
        </div>

        <script>
            Vue.component('child', {
                props: {
                    content: String
                },
                template: '<div>{{content}}</div>'
            })
            var vm = new Vue({
                el:'#app'
            })
        </script>
```
.....

上例中子组件通过props属性来接收父组件传递的数据, 但是与之前不同的是, props属性被设置为了对象而不是数组, 通过这种方式就能进行对接收到的props属性的类型验证了;
上例如果修改为:
```
.....
<child :content='123'></child>
.....
```

那么页面虽然还是能够正确显示(子组件还是接收了这个数字类型的props属性), 但是控制台会报错:
[Vue warn]: Invalid prop: type check failed for prop "content". Expected String, got Number.

当然, 这个类型检测可以接收多个值, 如:

```
.....
        props: {
                content: [String,Number]
            },
.....
```

例子2:

```
.....
        <div id='app'>
            <child></child>
        </div>

        <script>
            Vue.component('child', {
                props: {
                    content: {
                        type: String,
                        required: true,
                        default: 'default value'
                    }
                },
                template: '<div>{{content}}</div>'
            })
            var vm = new Vue({
                el:'#app'
            })
        </script>
.....
```

上例中在页面显示:
default value

并且在控制台报错:
[Vue warn]: Missing required prop: "content"

如果上例改为:
<child :content='123'></child>

那么页面显示:
123

并且在控制台报错:
[Vue warn]: Invalid prop: type check failed for prop "content". Expected String, got Number.

例子3:

```
.....
<div id='app'>
        <child :content="'abc'"></child>
    </div>

    <script>
        Vue.component('child', {
            props: {
                content: {
                    type: String,
                    validator: function(val){
                        return val.length > 5
                    }
                }
            },
            template: '<div>{{content}}</div>'
        })
        var vm = new Vue({
            el:'#app'
        })
    </script>
.....
```

上例在页面中显示为:
abc

并且控制台报错:
[Vue warn]: Invalid prop: custom validator check failed for prop "content".

Props特性和非Props特性;

上例中对content属性的使用就是属于Props特性, 因为它满足了一个条件:
父组件在子组件上定义了某个属性, 子组件本身通过props属性来接收了这个属性;
这种情况下在页面加载后html为:

```
<div id="app">
    <div>abc</div>
```

可以发现content属性并没有出现在渲染后的页面标签中, 这就是Props特性; 当然另一个Props特性是能够在子组件中使用接收到的props属性;

而非Props特性指的是当父组件在子组件上设置了一个属性, 但是子组件内部并没有通过props属性来接收这个属性, 那么此时就属于非Props特性, 这种情况下是无法在子组件中使用这个属性的, 并且页面加载后的html为:

```
<div id="app">
    <div content="abc"></div>
</div>
```

可以发现这个content属性被显示在了渲染后的页面标签中;

组件绑定原生事件;

例子1:

```
.....
        <div id='app'>
            <child @click="handleClick"></child>
        </div>

        <script>
            Vue.component('child', {
                template: '<div @click="handleClick">child</div>',
                methods:{
                    handleClick: function(){
                        console.log('child click')
                    }
                }
            })
            var vm = new Vue({
                el:'#app',
                methods: {
                    handleClick: function(){
                        console.log('click')
                    }
                }
            })
        </script>
```

.....

上例在页面加载后点击child组件会在控制台输出:
child click

上例说明在组件模板上直接定义的事件都属于自定义事件, 如上例中的:
<child @click="handleClick"></child>

这个click事件并非原生的click事件, 而被Vue理解为一个名为click的自定义事件, 需要子组件使用$emit()主动触发才能被父组件接收;

而在子组件中定义的click事件:
<div @click="handleClick">child

由于是在原生标签上定义的, 所以会被Vue解释为一个原生的click事件, 所以可以触发对应的handleClick方法输出:
child click

所以如果将上例子组件中的handleClick方法改为:

```
.....
        handleClick: function(){
                                        console.log('child click')
                                        this.$emit('click')
                                }
.....
```

那么在页面加载后点击child组件控制台输出:
child click
click

例子2:

```
.....
                <div id='app'>
                        <child @click.native="handleClick"></child>
                </div>

                <script>
                        Vue.component('child', {
                                template: '<div>child</div>'
                        })
                        var vm = new Vue({
```

```
                el:'#app',
                methods: {
                    handleClick: function(){
                        console.log('click')
                    }
                }
            })
        </script>
.....
```

上例中使用了@click.native这样的click事件绑定方式来告知Vue这个设置在组件元素上的事件是原生事件而非自定义事件, 所以就可以省去了像例子1中那样通过子组件向外emit一个自定义事件来让父组件执行点击事件回调函数这么繁琐的过程;

上例在页面加载后点击child组件在控制台显示:
click

非父子组件间的传值;

在Vue中, 如果涉及到了非父子组件之间的数据传递, 就需要使用Vuex框架或者Vue的总线(Bus)机制(发布/订阅模式);

例子:

```
.....
        <div id='app'>
            <child content="one"></child>
            <child content="two"></child>
        </div>

        <script>

            Vue.prototype.bus = new Vue();

            Vue.component('child', {
                props: {
                    content: String
                },
                data: function(){
                    return {
                        myContent: this.content
                    }
```

```
                },
                template: '<div @click="handleClick">{{myContent}}</div>',
                methods: {
                    handleClick: function(){
                        this.bus.$emit('change', this.myContent)
                    }
                },
                mounted: function(){
                    var _this = this
                    this.bus.$on('change', function(msg){
                        _this.myContent = msg
                    })
                }
            })
            var vm = new Vue({
                el:'#app'
            })
        </script>
.....
```

上例中使用了Vue.prototype.bus = new Vue() 语句在Vue这个类的原型上定义了一个bus属性, 这个属性指向一个统一的Vue实例; 之后初始化的任何Vue实例都可以获取到这个bus属性;

因为这个bus属性所有Vue实例都能够获取, 而在一个Vue实例上又可以通过$emit和$on方法来订阅/发布某个特定事件并传递数据, 所以这种方式就是解决Vue中非父子组件之间传值的好办法;

上例通过子组件的点击事件在bus实例对象上触发了一个change事件, 并传递了数据, 之后又在这个bus实例对象上监听了change事件来更改子组件自己的myContent属性;

所以当页面加载后点击某一个子组件时, 在bus这个实例对象上会先触发一个对应的change事件, 然后bus对象自己监听到了这个change事件于是触发了两个回调函数, 分别修改了两个子组件自己的myContent属性;
所以当点击one时, 页面中显示:
one
one

当点击two时, 页面中显示:
two
two

在Vue中使用插槽;

例子1:

```
.....
        <div id='app'>
            <child content="<p>a</p><p>b</p><p>c</p>"></child>
        </div>

        <script>

            Vue.component('child', {
                props:['content'],
                template: `<div>
                                <p>hello</p>
                                <div v-html="this.content"></div>
                                </div>`
            })
            var vm = new Vue({
                el:'#app'
            })
        </script>
.....
```

上例中想通过父元素传递给子元素一些需要在其模板中显示的html数据, 为了不被转义, 所以只能通过:

这样的形式来插入模板, 需要注意的是, 这里如果使用:
<template v-html="this.content"></template>
页面中不会渲染这个template标签中的内容;

那么上例中的这种方式虽然能在页面中显示正确的内容, 但是既不方便阅读和扩展, 并且结构上多了一个 元素, 所以需要更加好的解决方法;

Vue有一个插槽功能可以解决这类问题(类似react中this.props.children的机制);

例子2:

```
.....
        <div id='app'>
            <child>
                <p>world</p>
                <p>!</p>
```

```
            </child>
        </div>

        <script>

            Vue.component('child', {
                props:['content'],
                template: `<div>
                                <p>hello</p>
                                <slot></slot>
                            </div>`
            })
            var vm = new Vue({
                el:'#app'
            })
        </script>
.....
```

上例在页面中的html为:

```
<div id="app">
    <div>
        <p>hello</p>
        <p>world</p>
        <p>!</p>
    </div>
</div>
```

需要注意的是, <slot>标签中还可以指定默认值, 如:

```
.....
    <child>
    </child>
.....
<slot><h1>default</h1></slot>
.....
```

只有当child组件元素中没有写入任何内容的情况下这个默认值才会被渲染;

例子3:

```
.....
        <div id='app'>
            <body-content>
                <div class="header" slot="header">header</div>
```

```
                <div class="footer" slot="footer">footer</div>
            </bodu-content>
        </div>

        <script>

            Vue.component('body-content', {
                template: `<div>
                                <slot name="header"></slot>
                                <div class="content">content</div>
                                <slot name="footer"></slot>
                            </div>`
            })
            var vm = new Vue({
                el:'#app'
            })
        </script>
.....
```

上例中使用了'具名插槽'这个功能来实现了父组件传递多个插槽给子组件渲染的效果;
上例在页面中的html为:

```
<div>
    <div class="header">header</div>
    <div class="content">content</div>
    <div class="footer">footer</div>
</div>
```

当然, 具名插槽也可以设置默认值, 如:

```
<slot name="header">default header</slot>
```

那么当父元素没有传递一个带有属性: slot="header" 的元素时, 子组件中设置了name="header"的slot元素就会渲染为默认值;

需要注意的是, 上例中如果将'content'定义为子组件的名称会报错:
[Vue warn]: Do not use built-in or reserved HTML elements as component id: content

作用域插槽;

例子:

```
.....
        <div id='app'>
```

```
                <child>
                    <template slot-scope="props">
                        <h1>{{props.item}}</h1>
                    </template>
                </child>
        </div>

        <script>

            Vue.component('child', {
                data:function(){
                    return {
                        list:[1,2,3,4,5]
                    }
                },
                template: `<div>
                            <ul>
                                <slot
                                    v-for="item of list"
                                    :item=item
                                ></slot>
                            </ul>
                            </div>`
            })
            var vm = new Vue({
                el:'#app'
            })
        </script>
```
.....

上例在页面中的html为:

```
<div id="app">
    <div>
        <ul>
            <h1>1</h1>
            <h1>2</h1>
            <h1>3</h1>
            <h1>4</h1>
            <h1>5</h1>
        </ul>
    </div>
</div>
```

作用域插槽的作用其实就是:
当子组件中需要被渲染的内容一部分需要由父组件传递的插槽决定, 还有一部分需要由子组件自己的数据决定, 这种情况下就需要父组件定义所需传递给子组件的插槽的同时接收子组件传递的数据, 这样才能完整地展示插槽的内容;
上例中需要解决的问题就是:
子组件中需要循环展示其list数组中的内容, 但是用来循环展示内容的元素类型又需要父组件通过插槽来决定(上例中使用了h1, 但是也可以为li或其他指定值), 由于<slot>的机制所限定, 子组件中不可能通过类似: <slot>{{item}}</slot> 这样的形式来既接收了父组件传递的插槽元素, 又能将它自己的数据放入到插槽中展示, 所以就需要一个功能来让父元素结合子元素需要显示的数据来定义插槽内容;

需要注意的是, 上例中父元素中指定template元素并在其中指定slot-scope属性这种方式是固定格式, 相当于将子元素中指定的<slot>替换为了:
```
<template v-for="item of list">
        <h1>{{item}}</h1>
</template>
```

那么显然slot-scope这个属性所获取的就是子元素设置的<slot>的作用域了(它的所有属性), "props"这个值可以随意定义, 它代表的就是<slot>的作用域, 所以可以通过props.item获取子元素在<slot>上设置的:item=item这个属性的值;

动态组件和v-once指令;

例子1:
```
……
  <div id="app">
    <child-one v-if="type === 'child-one' "></child-one>
    <child-two v-if="type === 'child-two' "></child-two>
    <button @click="handleClick">change</button>
  </div>
  <script>
    Vue.component('child-one',{
      template: '<div>child-one</div>'
    })
    Vue.component('child-two',{
      template: '<div>child-two</div>'
    })
    var vm = new Vue({
      el: '#app',
      data:{
        type:'child-one'
      },
```

```
        methods:{
          handleClick: function(){
            this.type = this.type === 'child-one' ? 'child-two' : 'child-one'
          }
        }
      })
    </script>
……
```

上例中实现了一个toggle子组件的功能;
在页面加载后显示: child-one
当点击change按钮后页面中显示: child-two

例子2:

```
……
<div id="app">
        <component :is="type"></component>
        <button @click="handleClick">change</button>
</div>
……
```

例子2与例子1在页面中的效果完全相同;
component是Vue的一个内置组件, 称为动态组件, 它通过is属性来决定哪个组件被渲染;

参考:
https://cn.vuejs.org/v2/api/#component

例子3:

```
……
        Vue.component('child-one',{
          template: '<div v-once>child-one</div>'
        })
        Vue.component('child-two',{
          template: '<div v-once>child-two</div>'
        })
……
```

上例中在两个子组件中添加了v-once指令(页面中效果与例子1,2相同), 它的作用是: 一旦组件被渲染后就会被缓存在内存中, 如果之后组件被销毁后重新创建就会直接从内存中获取已经保存的内容, 而无需重新创建, 能够在一定程度上提高性能;
需要注意的是, 设置了v-once指令的元素和组件只会被渲染**一次**, 随后的重新渲染, 元

素/组件及其所有的子节点将被视为静态内容并跳过; 这可以用于优化更新性能;

参考:
https://cn.vuejs.org/v2/api/#v-once

v-once指令与Vue的内置组件keep-alive作用类似(主要用于保留组件状态或避免重新渲染), 参考:
https://cn.vuejs.org/v2/api/?#keep-alive

Vue中的CSS动画原理(实现过渡效果);

Vue的transition组件机制类似react的react-addons-css-transition-group组件(其实它的功能与之后要提到的transition-group更类似);

例子:

```
.....
    <style>
      .fade-enter{
        opacity: 0;
      }
      .fade-enter-active{
        transition: opacity 2s;
      }
      .fade-leave-to {
        opacity: 0;
      }
      .fade-leave-active{
        transition: opacity 2s;
      }
    </style>
  </head>
  <body>
    <noscript>
      You need to enable JavaScript to run this app.
    </noscript>
    <div id="app">
      <transition name="fade">
        <div v-if="show">hello world</div>
      </transition>
      <button @click="handleClick">toggle</button>
    </div>
```

```
    <script>
        var vm = new Vue({
          el: '#app',
          data:{
            show: true
          },
          methods:{
            handleClick: function(){
              this.show = !this.show
            }
          }
        })
    </script>
```
……

上例中, Vue会分析被<transition name=“fade”>组件包裹元素的状态, 并且执行以下动作:
<1>当元素被渲染时;
在动画开始执行时(动画第一帧), Vue会为被包裹元素添加: fade-enter和fade-enter-active这两个样式;
当动画第一帧结束后, Vue会在动画的第二帧中把元素上的fade-enter样式移除, 并添加一个fade-enter-to样式;
当动画结束时(最后一帧), Vue会移除元素上剩下的两个样式: fade-enter-to和fade-enter-active;
<2>当元素被移除时;
在动画开始执行时(动画第一帧), Vue会为被包裹元素添加: fade-leave和fade-leave-active这两个样式;
当动画第一帧结束后, Vue会在动画的第二帧中把元素上的fade-leave样式移除, 并添加一个fade-leave-to样式;
当动画结束时(最后一帧), Vue会移除元素上剩下的两个样式: fade-leave-to和fade-leave-active;

其实就是利用css3的transition样式属性来监听opacity的变化(过渡动画效果), 一旦有变化了就根据指定的时长执行动画效果;

所以上例中设置的样式能够让被<transition name=“fade”>组件包裹的 元素实现渐入/渐出效果;

注意:
1.上例中在<transition>中指定的name="fade”决定了Vue将会为其中元素添加的样式名称为类似: fade-enter这样的形式, 如果不指定name属性, 那么Vue默认会使用的样式是类似: v-enter, v-leave-active这样的形式;
2.在<transition>中还可以自定义Vue将会添加的enter/leave样式的名称格式, 如:

```
……
    <style>
      .fade-enter{
        opacity: 0;
      }
      .enter{
        transition: opacity 2s;
      }
      .fade-leave-to {
        opacity: 0;
      }
      .leave{
        transition: opacity 2s;
      }
    </style>
  </head>
  <body>
    <noscript>
      You need to enable JavaScript to run this app.
    </noscript>
    <div id="app">
      <transition name="fade"
                  enter-active-class="enter"
                  leave-active-class="leave">
        <div v-if="show">hello world</div>
      </transition>
      <button @click="handleClick">toggle</button>
    </div>
……
```

上例在页面的效果与之前的例子相同;

3.上例中如果将被<transition name=”fade”>组件包裹的元素中v-if指令改为v-show指令, 动画效果仍旧生效; 另外, 在<transition>中使用动态组件也同样能够展示对应效果;

在Vue中使用animate.css库(实现动画效果);

例子1:

```
……    <style>
      @keyframes bounce-in {
        0% {
```

```
        transform: scale(0);
      }
      50% {
        transform: scale(1.5);
      }
      100% {
        transform: scale(1);
      }
    }
    .fade-enter-active{
      transform-origin: left center;
      animation: bounce-in 1s;
    }
    .fade-leave-active {
      transform-origin: left center;
      animation: bounce-in 1s reverse;
    }
  </style>
</head>
<body>
  <noscript>
    You need to enable JavaScript to run this app.
  </noscript>
  <div id="app">
    <transition name="fade">
      <div v-if="show">hello world</div>
    </transition>
    <button @click="handleClick">toggle</button>
  </div>
  <script>
      var vm = new Vue({
        el: '#app',
        data:{
          show: true
        },
        methods:{
          handleClick: function(){
            this.show = !this.show
          }
        }
      })
  </script>
……
```

上例中使用了Vue的transition组件配合CSS3的@keyframes/animation来实现了元素放大/缩小的动画效果;

例子2:

下载animate.css后引入index.html

```
……
   <link rel="stylesheet" type="text/css" href="./animate.css"/>
    <style>
    </style>
  </head>
  <body>
    <noscript>
      You need to enable JavaScript to run this app.
    </noscript>
    <div id="app">
      <transition
        enter-active-class="animated swing"
        leave-active-class="animated shake"
      >
        <div v-if="show">hello world</div>
      </transition>
      <button @click="handleClick">toggle</button>
    </div>
    <script>
        var vm = new Vue({
          el: '#app',
          data:{
            show: true
          },
          methods:{
            handleClick: function(){
              this.show = !this.show
            }
          }
        })
    </script>
……
```

上例中使用了Vue的transition组件配合animate.css库来实现指定元素入场/出场的动画效果;

需要注意的是, 如果要让Vue的transition组件使用animate.css库, 就必须使用自定义样式名(enter-active-class/leave-active-class)的方式, 并且样式名需要以animated开头, 后面跟需要使用的animate.css中的动画名(https://daneden.github.io/animate.css/ 首页可以查看并测试所有动画效果);
其实transition组件上的enter-active-class/leave-active-class属性就是用来设置在被包裹元素出/入场时实际添加/删除的类名, 上例中的设置方法其实就是在元素入场时为其添加/删除样式类:
class='animated swing'
在其出场时为其添加/删除样式类:
class='animated shake'

而这两个类指定的动画样式可以在animate.css源码中查看;

关于css3中animation的用法, 参考:
http://www.css88.com/book/css/properties/animation/animation.htm (重要)
https://developer.mozilla.org/zh-CN/docs/Web/CSS/transform-origin (重要, transform-origin用法)

animate.css
https://daneden.github.io/animate.css/ (官网, API)

上例中页面加载后并不会展示任何动画效果, 只有在用户点击toggle按钮后才会显示动画效果, 如果这里需要页面在加载后就让div元素展示入场的动画效果就需要设置:

```
……
    <transition
      appear
      appear-active-class="animated swing"
      enter-active-class="animated swing"
      leave-active-class="animated shake"
    >
……
```

需要注意的是, appear这个属性必须设置, 它告诉transition组件开启对其包裹的组件首次渲染时的动画展示功能, 并且通过appear-active-class属性指定了动画类型;

例子3:

```
……
  <link rel="stylesheet" type="text/css" href="./animate.css"/>
  <style>
    .fade-enter, .fade-leave-to{
      opacity: 0;
```

```
    }
    .enter,
    .leave{
      transition: opacity 2s;
    }
  </style>
</head>
<body>
  <noscript>
    You need to enable JavaScript to run this app.
  </noscript>
  <div id="app">
    <transition
      name="fade"
      appear
      appear-active-class="animated swing"
      enter-active-class="animated swing enter"
      leave-active-class="animated shake leave"
    >
      <div v-if="show">hello world</div>
    </transition>
    <button @click="handleClick">toggle</button>
  </div>
  <script>
      var vm = new Vue({
        el: '#app',
        data:{
          show: true
        },
        methods:{
          handleClick: function(){
            this.show = !this.show
          }
        }
      })
  </script>
……
```

上例中通过:
enter-active-class="animated swing enter"
leave-active-class="animated shake leave"

让transition组件通过animate.css中的动画样式为包裹元素设置了入场/出场动画, 同时

又自定义了名为enter/leave的两个出/入场样式, 让transition组件在动画展示生命周期中添加/删除这两个样式, 这样用户就可以在使用animate.css库的同时自定义一些出/入场效果了;

上例中被transition组件包裹元素上同时会展现animate.css和用户自定义的动画效果;

这里会存在一个问题, 因为Vue的transition组件是以其v-enter-active/v-leave-active中设置的动画时长做为依据来决定动画效果的生命周期的(经过多次时间才会移除所有相关样式), 那么上例中自定义的enter/leave样式类中指定的动画时长为2s, 而animate.css中在.animated类里设置的动画时长默认为1s:

```
.....
.animated {
  -webkit-animation-duration: 1s;
  animation-duration: 1s;
  -webkit-animation-fill-mode: both;
  animation-fill-mode: both;
}
.....
```

此时transition组件就可能会无法正确判断动画效果的具体生命周期长度, 所以需要手动指定:

```
.....
<transition
        type="transition"
        name="fade"
        appear
        appear-active-class="animated swing"
        enter-active-class="animated swing enter"
        leave-active-class="animated shake leave"
      >
.....
```

上例中通过在transition组件上指定type="transition”来告诉Vue以transition这个样式属性设置的时长为准;

当然也可以指定具体的时长, 如:

```
.....
<transition
        :duration="5000"
        name="fade"
        appear
        appear-active-class="animated swing"
```

```
      enter-active-class="animated swing enter"
      leave-active-class="animated shake leave"
    >
……
```

或者

```
……
<transition
      :duration="{enter:3000, leave:5000}"
      name="fade"
      appear
      appear-active-class="animated swing"
      enter-active-class="animated swing enter"
      leave-active-class="animated shake leave"
    >
……
```

需要注意的是, 上例中指定的是transition组件所包裹元素的动画效果生命周期持续长度, 也就是说, 用户之前指定的animate.css的动画样式依旧会在1s后结束, 通过transition过渡样式属性指定的动画效果依旧会在2s后结束, 但是transition组件对元素上相关样式的移除会在用户指定的duration后才执行(被transition组件所管理的元素/组件的移除动作也会因为动画生命周期的长度而推迟);

Vue中的JS动画与Velocity.js的结合使用;

Vue的transition组件除了能够借助为包裹元素添加/删除css样式来实现动画效果, 还提供了一些js动画钩子;

例子1:

```
……
  <div id="app">
    <transition
      name="fade"
      @before-enter="handleBeforeEnter"
      @enter="handleEnter"
      @after-enter="handleAfterEnter"
    >
      <div v-show="show">hello world</div>
    </transition>
    <button @click="handleClick">toggle</button>
  </div>
```

```
    <script>
      var vm = new Vue({
        el: '#app',
        data:{
          show: true
        },
        methods:{
          handleClick: function(){
            this.show = !this.show
          },
          handleBeforeEnter: function(el){
            el.style.color = 'red'
          },
          handleEnter: function(el, done){
            setTimeout(()=>{
              el.style.color='green'
            },2000)
            setTimeout(()=>{
              done()
            },3000)
          },
          handleAfterEnter: function(el){
            el.style.color="#000"
          }
        }
      })
    </script>
……
```

上例中在transition组件上设置了3种js动画钩子函数:
before-enter
enter
after-enter

需要注意的是, 这三个钩子函数都会将被包裹元素的元素对象做为第一个参数传入函数, 而enter函数会接收第二个参数, 这个参数是一个函数, 只有当这个函数被执行时transition组件才会认为enter动画已经执行完毕, 才会去触发after-enter这个钩子;

所以上例在页面中的效果是:
当用户第二次点击toggle按钮时hello world重新显示在页面上, 并且字体为红色, 过了2s后字体变为绿色, 又过了1s后字体变回黑色;

例子2:

```
……
    <div id="app">
      <transition
        name="fade"
        @before-leave="handleBeforeLeave"
        @leave="handleLeave"
        @after-leave="handleAfterLeave"
      >
        <div v-show="show">hello world</div>
      </transition>
      <button @click="handleClick">toggle</button>
    </div>
    <script>
        var vm = new Vue({
          el: '#app',
          data:{
            show: true
          },
          methods:{
            handleClick: function(){
              this.show = !this.show
            },
            handleBeforeLeave: function(el){
              el.style.color = 'red'
            },
            handleLeave: function(el, done){
              setTimeout(()=>{
                el.style.color='green'
              },2000)
              setTimeout(()=>{
                done()
              },3000)
            },
            handleAfterLeave: function(el){
              el.style.color="#000"
            }
          }
        })
    </script>
……
```

上例中使用了leave动画的3种js动画钩子:

before-leave
leave
after-leave

它们的机制与enter动画的3个钩子对应相同;

所以上例在页面中的效果为:
页面中显示hello world, 字体为黑色, 当用户第一次点击toggle按钮时字体变为红色, 2s后字体变为绿色, 又过了1s后hello world消失, 当用户再次点击toggle按钮后hello world重新显示, 字体为黑色;

例子3:

下载velocity.js;

```
.....
    <script type="text/javascript" src="./velocity.js"></script>
  </head>
  <body>
    <noscript>
      You need to enable JavaScript to run this app.
    </noscript>
    <div id="app">
      <transition
        name="fade"
        @before-enter="handleBeforeEnter"
        @enter="handleEnter"
        @after-enter="handleAfterEnter"
      >
        <div v-show="show">hello world</div>
      </transition>
      <button @click="handleClick">toggle</button>
    </div>
    <script>
       var vm = new Vue({
         el: '#app',
         data:{
           show: true
         },
         methods:{
           handleClick: function(){
             this.show = !this.show
           },
```

```
        handleBeforeEnter: function(el){
          el.style.opacity = 0;
        },
        handleEnter: function(el, done){
          Velocity(el, {opacity:1}, {duration: 1000, complete: done})
        },
        handleAfterEnter: function(el){
          Velocity(el, {opacity:0.5, background:'grey'}, {duration: 1000})
        }
      }
    })
  </script>
……
```

上例中结合了Vue的transition组件和Velocity.js来实现了元素入场的动画效果;

补充:
1.velocity.js配合jQuery使用;
replace all instances of jQuery's $.animate() with $.velocity(). You will immediately see a performance boost across all browsers and devices — especially on mobile.

参考:
http://velocityjs.org/ (官网)
http://www.mrfront.com/docs/velocity.js/index.html (中文文档)

Vue中多个元素/组件的过渡效果;

注意:
这里的多个元素指的是它们之间存在互相切换的效果, 也就是说同一时刻只会有一个元素/组件存在于transition组件中;
transition组件只能处理一个单一的元素/组件(同一时间只能处理一个动画生命周期), 所以只有当多个元素/组件存在互相切换的条件时才能在transition组件中设置多个元素/组件;

例子1:

```
……
  <style>
    .v-enter,.v-leave-to {
      opacity: 0;
    }
    .v-enter-active, .v-leave-active {
```

```
      transition: opacity 1s;
    }
  </style>
 </head>
 <body>
  <noscript>
   You need to enable JavaScript to run this app.
  </noscript>
  <div id="app">
   <transition mode='out-in'>
    <div v-if="show" key="hello">hello world</div>
    <div v-else key="bye">bye</div>
   </transition>
   <button @click="handleClick">toggle</button>
  </div>
  <script>
    var vm = new Vue({
     el: '#app',
     data:{
      show: true
     },
     methods:{
      handleClick: function(){
       this.show = !this.show
      }
     }
    })
  </script>
……
```

上例中实现的效果是:
页面加载后显示hello world, 当用户第一次点击toggle按钮后hello world逐渐隐藏, bye逐渐显示, 完成交替; 当用户再次点击toggle按钮后, bye逐渐隐藏, hello world逐渐显示, 完成交替;

需要注意的是, 上例中如果不在transition组件包裹的div元素上设置key属性, 那么就不会展示过渡效果, 因为Vue在更新组件时会尽力复用模板中没有改变的部分, 所以只会更新页面中div元素的内容, 而不会移除/重新渲染这个div元素, 而transition组件是依据其所包裹元素/组件是否被移除/重新渲染(或者根据display样式属性进行显示/隐藏)来判断是否启动入场/出场的动画生命周期的, 所以上例中需要给被包裹的两个div元素添加key属性来让Vue在组件更新时移除/重新渲染div元素来开启动画生命周期;

上例中在transition组件中设置了mode属性, 这个属性用来设置当同时存在入场/出场动

画效果需要被展示时transition组件处理的规则;
上例中的mode='out-in'的作用是: 让出场动画先执行, 完成后再展示入场动画;
如果mode='in-out', 那么入场动画会先执行, 完成后在展示出场动画;
需要注意的是, 这里的出场动画生命周期执行完毕指的是: 需要被移除/隐藏的元素/组件上所有出场相关样式类被移除, 并且这个元素/组件被从页面上真正移除后再执行入场动画生命周期(包括将元素/组件添加到页面上这个动作);

例子2:

```
.....
    <style>
      .v-enter,.v-leave-to {
        opacity: 0;
      }
      .v-enter-active, .v-leave-active {
        transition: opacity 1s;
      }
    </style>
  </head>
  <body>
    <noscript>
      You need to enable JavaScript to run this app.
    </noscript>
    <div id="app">
      <transition mode='out-in'>
        <component :is='type'></component>
      </transition>
      <button @click="handleClick">toggle</button>
    </div>
    <script>

        Vue.component('child-one',{
          template: '<div>child one</div>'
        })
        Vue.component('child-two',{
          template: '<div>child two</div>'
        })
        var vm = new Vue({
          el: '#app',
          data:{
            type: 'child-one'
          },
          methods:{
```

```
        handleClick: function(){
          this.type = this.type === 'child-one'?'child-two':'child-one'
        }
      }
    })
  </script>
```
.....

上例中使用了动态组件配合transition组件实现了组件之间切换显示的过渡效果(与例子1中切换效果类似);
需要注意的是, 如果是动态组件之间的切换, 那么Vue将一定会移除/重新渲染对应组件, 而不会去检查复用任何组件的template内容, 所以无需指定key属性;

Vue中的列表过渡;

例子:
.....
```
  <style>
    .v-enter,.v-leave-to {
      opacity: 0;
    }
    .v-enter-active, .v-leave-active {
      transition: opacity 1s;
    }
  </style>
 </head>
 <body>
  <noscript>
    You need to enable JavaScript to run this app.
  </noscript>
  <div id="app">
    <transition-group>
      <div v-for="item of list" :key="item.id">{{item.title}}</div>
    </transition-group>
    <button @click="handleClick">add</button>
  </div>
  <script>

    var count = 0;

    var vm = new Vue({
```

```
        el: '#app',
        data:{
          list: []
        },
        methods:{
          handleClick: function(){
            this.list.push({id:count++, title:'hello world'})
          }
        }
      })
    </script>
……
```

上例中使用了transition-group组件来实现对多个元素/组件实现过渡动画的效果, 之前提到过transition组件只能处理单一的元素/组件的动画效果, 所以如果上例中将transition-group标签改为transition, 会报错:
[Vue warn]: <transition> can only be used on a single element. Use <transition-group> for lists.

上例实现的效果是:
当用户点击add按钮后就会在页面中添加一项内容为hello world的div元素, 并且存在入场动画效果;

其实transition-group的原理是将其中包裹的多个单一元素用多个transition组件包裹起来, 如:

```
<transition-group>
      <div>1</div>
      <div>2</div>
      <div>3</div>
</transition-group>
```

相当于:

```
<transition>
      <div>1</div>
</transition>
<transition>
      <div>2</div>
</transition>
<transition>
      <div>3</div>
</transition>
```

Vue中的动画封装(可复用的动画效果);

例子:

```
.....
  <div id="app">
    <fade :show='show'>
      <h1>hello world</h1>
    </fade>
    <button @click="handleClick">toggle</button>
  </div>
  <script>

    Vue.component('fade', {
      props: ['show'],
      template: `
        <transition
          @before-enter='handleBeforeEnter'
          @enter='handleEnter'>
          <slot v-if='show'></slot>
        </transition>
      `,
      methods: {
        handleBeforeEnter: function(el){
          el.style.color = 'red'
        },
        handleEnter: function(el, done){
          setTimeout(()=>{
            el.style.color = 'green'
            done()
          },2000)
        }
      }
    })

    var vm = new Vue({
      el: '#app',
      data:{
        show: true
      },
      methods:{
        handleClick: function(){
          this.show = !this.show
```

```
        }
      }
    })
  </script>
```
……

上例中使用了fade组件封装了一个入场的动画效果, 可以直接调用这个组件来实现相关过渡效果, 使用方法是: 为fade组件传入一个名为show的属性来指定需要动画效果的元素的显示状态, 并且以slot的形式传入需要实现动画效果的元素即可;
需要注意的是, 在<slot>标签中只能通过v-if这个条件指令来决定插槽对应的元素是否被渲染, v-show指令无效(不会添加display样式属性);

Vue中状态过渡动画可以参考(数据变化的过渡动画):
https://cn.vuejs.org/v2/guide/transitioning-state.html

……