Vue

BV1Zy4y1K7SH P136

Vue2 完毕

初步使用

挂载元素的方式

第一种：`el:'#app'`，第二种：`vm.$mount('#app')`

data的两种挂载方式：对象式和函数式。

函数式：

```
data:function () {
      return{
            msg: "good",
            url: "https://baidu.com"
      }
}
```

vm上的数据代理类似`Object.defineProperty`，并且设置了`getter`和`setter`

vue的v-xxx

`v-bind`单向绑定数据可以简写为`:`，`v-model`双向绑定数据

```
<div id="app">
      <a :href="url">OK</a><br>
      <input type="text" v-model="msg"/>  {{msg}}
</div>
<script>
      new Vue({
            el:'#app',
            data:{
                  url: "https://www.baidu.com",
                  msg: "Jesus"
            }
      })
</script>
```

`v-model`限制符：限制数字：`v-model.number`，去除空格`v-model.trim`，懒加载`v-model.lazy`

`v-on`绑定事件，可以简写为`@`：

默认传入第一个参数为`event`参数，如果指定可以`dost($event, ...)`

```
<div id="app">
      <button @click="dost">button</button>
</div>
<script>
      new Vue({
            el: '#app',
            data: {
                  msg: "good",
            },
            methods: {
                  dost(e) {
                        alert("Nice try" + this.msg)
                  }
            }
      })
</script>
```

事件修饰符：

- `prevent`：`<a :href="url" @click.prevent="dost">OK</a><br>`可以防止a标签跳转网页
- `stop`： 阻止事件冒泡
- `once`： 事件只触发一次
- `self`： 只有`e.target`是本身才可以出发，某种程度上可以阻止冒泡。
- `passive`： 事件的默认行为会立即执行。

键盘事件：

`@keydown` `@keyup`

`@keydown.enter`按下enter键，vue中自定义的键`esc` `tab` `delete` `space` `left`

支持三元表达式：

```
<p :text="flag?'ok':'no'"></p>
```

`v-text` `v-html`

```
<p v-text="txt"></p>
<p v-html="txt"></p>
```

`v-cloak`

```
[v-cloak]{
      display: none;
}
```

`v-once`

只渲染一次，之后视为静态。

`v-pre`

跳过vue渲染的元素。

vue属性

计算属性`computed`

计算属性通过get方法return得到对应的值。

根据vm中属性计算而来的属性，这里的get在vm刷新后只读取一次。

> 对于计算属性的修改需要重写get和set方法。

```
<div id="app">
     A:<input type="text" v-model="a"><br>
     B:<input type="text" v-model="b"><br>
     {{c}}
</div>
<script>
      new Vue({
            el: '#app',
            data: {
                  a: "qwq",
                  b: "qaq"
           },
            computed:{
                  c:{
```

```
                        get(){
                                console.log("Invoke getter");
                                return this.a.slice(0,3) + '-' + this.b.slice(0,3);
                        }
                },
                 d(){      // 简写形式
                        return this.a.slice(0,3) + '+' + this.b.slice(0,3);
                }
        }
    })
</script>
```

监视属性 watch

会获取到监视变量改变前后的值

```
<div id="app">
      <input type="text" v-model="msg"/> {{msg}}
</div>
<script>
      const vm = new Vue({
            el: '#app',
            data: {
                  msg: "good",
            },
            watch: {
                  msg: {
                        immediate: true,    // 初始化立即调用handler
                        handler(newValue, oldValue) {
                              console.log(newValue, oldValue)
                        }
                  }
            }
      })
      // 或者
      vm.$watch('msg',{
            immediate: true,    // 初始化立即调用handler
            handler(newValue, oldValue) {
                  console.log(newValue, oldValue)
            })
</script>
```

深度监视：对于对象内部的变量监视使用字符串："dog.age"，可以监视多级结构，使用 deep:true 可以监视对象中的任意字段。

```
watch: {
      msg: {
            immediate: true,
            deep: true,
            handler(newValue, oldValue) {
                  console.log(newValue, oldValue)
            }
      }
}
```

样式绑定

```
<div class="good" :class="msg"></div>
多个样式
<div class="good" :class="['one','two']"></div>
对象写法
<div class="good" :class="{one:true, two:false}"></div>
也可以使用style内联样式
```

条件渲染 `v-if` `v-else` `v-else-if` `v-show`

多个if else的元素必须紧紧相连

v-if直接添加删除元素，v-show设置display属性

列表渲染 `v-for`

> v-for可以便利数组、对象、字符串

```
<div id="app">
   <ul>
      <li v-for="(d, index) in dogs" :key="d.id">{{index}}:{{d.name}}-{{d.age}}</li>
   </ul>
</div>
<script>
      new Vue({
            el: '#app',
            data: {
               dogs:[
                 {id:1, name:'Julia', age: 2},
                 {id:2, name:'Johny', age: 0.5},
                 {id:3, name:'Bass', age: 4},
              ]
            }
     })
</script>
```

key的作用：用于标识每个组内元素，防止错乱，一般不用index作为key。使用 `unshift` 的时候，会使用diff算法，导致某些bug。

列表过滤：

> 可以通过使用 `watch` 或者 `computed` 计算属性

```
watch:{
   keyword(val){
      this.aDogs = this.dogs.filter((d)=>{
         return d.name.indexOf(val) > -1
      })
  }
}
```

列表排序：

```
<div id="app">
   <input type="text" v-model="keyword" placeholder="Input sth">
   <button @click="sortType = 1">up Order</button>
   <button @click="sortType = 2">down Order</button>
   <button @click="sortType = 0">origin Order</button>
```

```
    <ul>
      <li v-for="(d, index) in aDogs" :key="d.id">{{d.name}}-{{d.age}}== {{d.sex}}</li>
    </ul>
</div>
<script>
    new Vue({
      el: '#app',
      data: {
        keyword: '',
        sortType: 0,
        dogs:[
          {id:1, name:'Julia', age: 2, sex: '男'},
          {id:2, name:'Johny', age: 0.5, sex: '男'},
          {id:3, name:'Bass', age: 4, sex: '女'},
          {id:4, name:'Burro', age: 4, sex: '女'},
        ]
      },
      computed:{
        aDogs(){
          let arr = this.dogs.filter((d)=>{
            return   d.name.indexOf(this.keyword) > -1
          })
          if (this.sortType){
            return arr.sort((a,b)=>{
              return this.sortType === 1? a.age-b.age:b.age-a.age;
            })
          }
          return arr;
        }
      }
    })
</script>
```

列表更新存在的问题：

具体参考：**列表渲染**

> 对于数组中的对象更新的时候，Vue会更新不到的。因为数组元素不具有 `getters and setters`，不能使用赋值的形式改变，只能使用数组的函数 `push, pop, shift, unshift` 来进行更改。

可以使用 `Vue.set()` 来添加对象的属性，在数据代理的时候自动添加 `getters and setters`。但是只适用于给 `data` 里中的对象类型添加属性。

vue收集表单数据

获取表单数据

```
<div id="app">
      <form>
           账户：<input type="text" v-model="account"><br>
           密码：<input type="text"><br>
           性别：男 <input type="radio" name="gender" value="male" v-model="sex"> 女 <input type="radio"
name="gender"

                      value="female" v-model="sex"><br>
           爱好：<input type="checkbox" name="hobby" value="smoke" v-model="hobby">抽烟<input
type="checkbox" name="hobby"
```

```
                        v-model="hobby"

                        value="drink">喝酒<input
                type="checkbox"
                name="hobby" v-model="hobby" value="tang">烫头<br>
            所属：<select name="school" v-model="school">
             <option value="A">A</option>
             <option value="A">B</option>
             <option value="A">C</option>
             <option value="A">D</option>
        </select><br>
            其他：<br><textarea name="" cols="30" rows="10"></textarea>
             <br><input type="checkbox" name="recv">接受
             <br>
             <button>OK</button>
        </form>
</div>
<script>
        new Vue({
              el: '#app',
              data: {
                    account: '',
                    passwd: '',
                    sex: 'male',
                    hobby: [],
                    school: 'A'
              }
        })
</script>
```

checkbox对于的数据需要为数组类型 `[]` ，对于radio类型的 `v-model` 需要指定value值。

对输入框进行限制，比如数字类型：

```
<input type="number" v-model.number="age"/>
```

过滤器

```
<div id="app">
     {{msg | rmblank}}
</div>
<script>
      new Vue({
            el: '#app',
            data: {
                  msg: 'I am a good boy'
            },
            filters:{
                  rmblank(v){
                        return v.replaceAll(' ','')
                  }
            }
      })
</script>
```

过滤器函数第一个参数始终为管道符前的参数。支持多个管道符串联，运算顺序从左到右。

上述为局部过滤器。

全局过滤器：

```
Vue.filter('rmblank2', function (v){
      return v.replaceAll(' ','')
})
```

自定义指令

函数式：

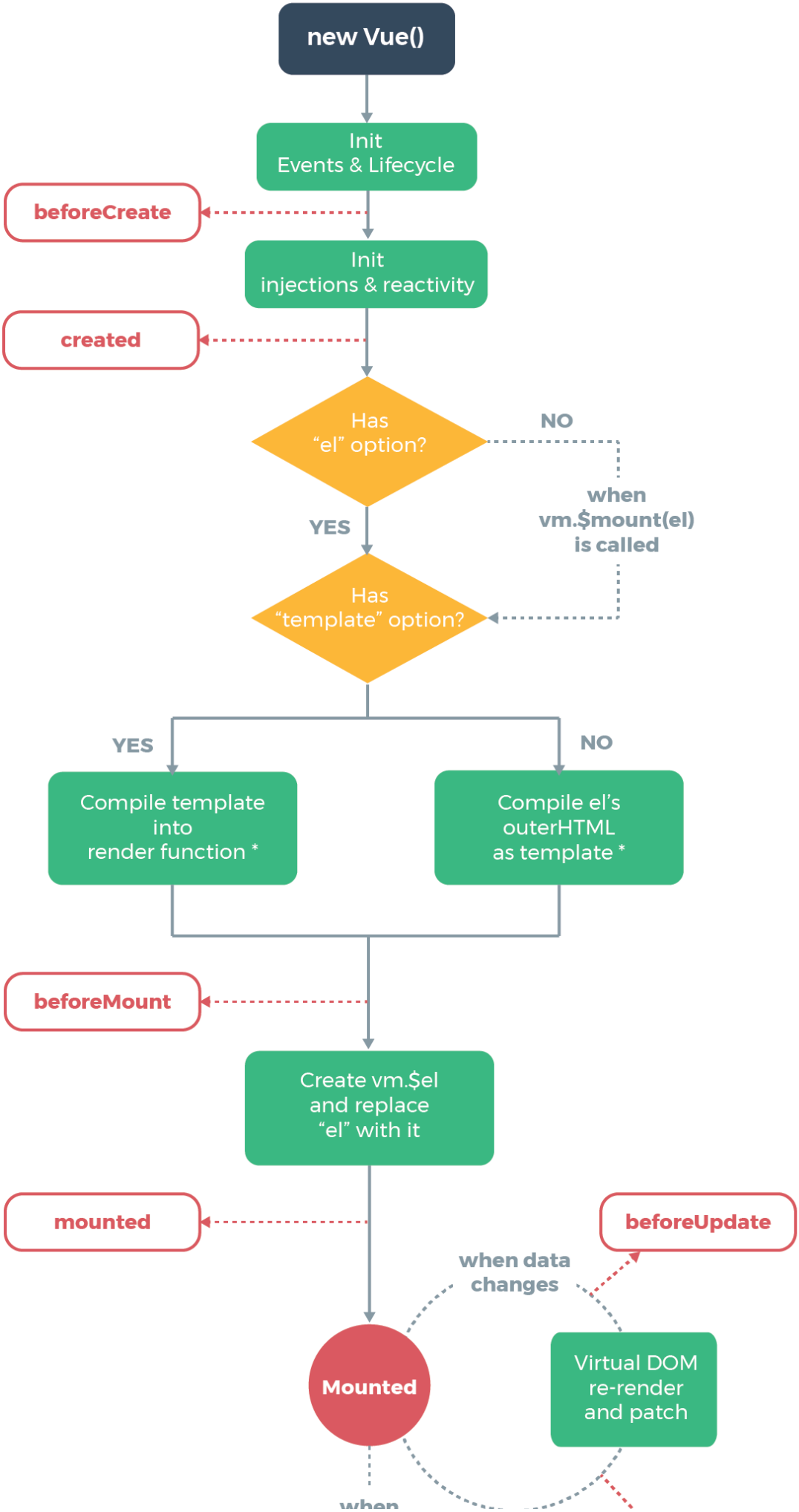
```
<div id="app">
      <p v-big="n"></p>
</div>
<script>
      new Vue({
            el: '#app',
            data: {
                  n: 1
            },
            directives:{
                  big(ele, binding){   // ele获取的是DOM元素，binding获取的是绑定关系
                        ele.innerText = binding.value * 10
                  }
            }
      })
</script>
```

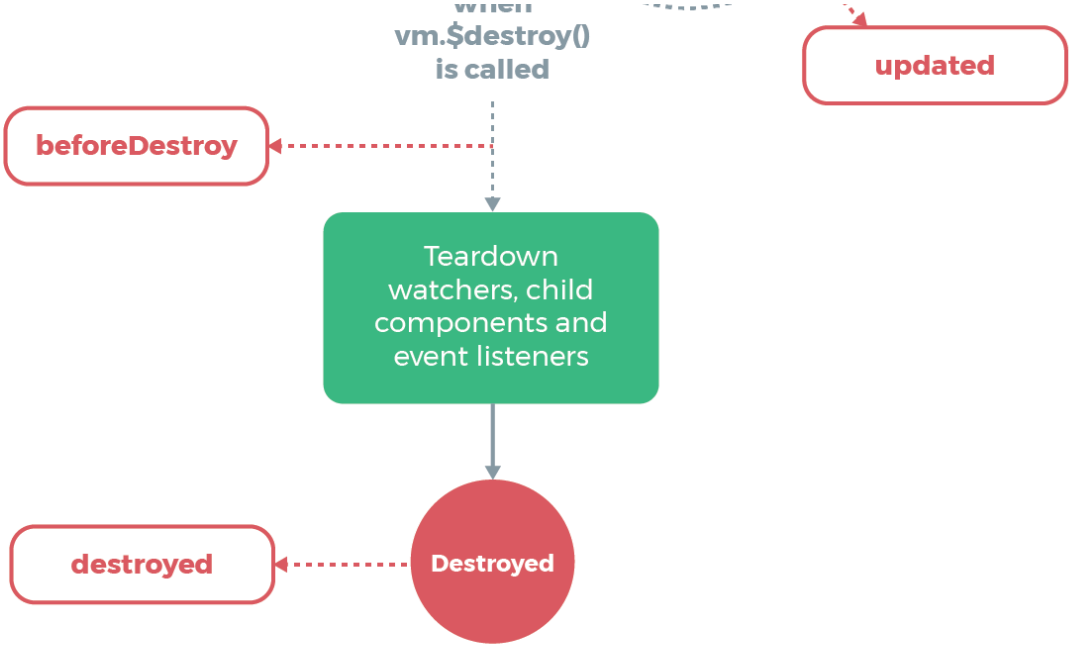
对象式：

```
<script>
      new Vue({
            el: '#app',
            data: {
                  n: 1
            },
            directives:{
                  fbind:{     // 对象式
                        bind(ele, binding){
                              // 指令与元素成功绑定的时候
                        },
                        inserted(ele, binding){
                              // 指令被插入到DOM中的时候
                        },
                        updated(ele, binding){
                              // 模板被更新的时候调用
                        }
                  }
            }
      })
</script>
```

生命周期

Vue 生命周期


new Vue()
Init
Events & Lifecycle
beforeCreate
Init
injections & reactivity
created
Has
“el” option?
NO
when
vm.$mount(el)
is called
YES
Has
“template” option?
YES
NO
Compile template
into
render function *
Compile el’s
outerHTML
as template *
beforeMount
Create vm.$el
and replace
“el” with it
mounted
beforeUpdate
when data
changes
Mounted
Virtual DOM
re-render
and patch
when



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

`mounted()` 之后是经过Vue编译后的界面。

`beforeUpdate`，数据是新的，页面是旧的

组件化编程

可以提高程序的复用性，相同组件之间不会互相影响。

非单文件使用法：

> `data` 必须使用函数形式定义，使用组件标签 `<dog></dog>`，标签支持嵌套。

```
<div id="app">
      <dog></dog>
</div>
<script>
      const dog = Vue.extend({
            template: `<h1>{{msg}}</h1>`,
            data(){
                  return{
                        msg: 'Hello, Component.'
                  }
            }
      })
      new Vue({
            el: '#app',
            components:{
                  dog
            }
```

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

单文件组件开发：

index.html

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

School.vue

```
<template>
   <div>
      <h3>{{ msg }}</h3>
   </div>
</template>

<script>
export default {
   name: 'School',
   data() {
      return{
         msg: 'Sole Component'
      }
  },
}
</script>
```

App.vue

```
<template>
   <div>
      <School></School>
   </div>
</template>

<script>
import School from "./School";
export default {
   name: "App",
   components: {
      School
  }
}
</script>
```

main.js

```
import Vue from "vue";
import App from "./App";

new Vue({
      el: '#app',
      render: h=>h(App)
})
```

浏览器不能直接识别ES6语法，需要使用到**Vue CLI**。

需要安装提前npm（nodejs）

```
# 配置淘宝镜像
npm config set registry https://registry.npm.taobao.org
# 安装VUE CLi
npm install -g @vue/cli
# 创建一个工程
vue create clidemo
# 开启网页
vue run serve
```

这里使用的Vue使用的是`vue.runtime.esm.js`，没有模板解析器。

ref属性

可以操作DOM元素

```
<template>
   <div>
      <h1 ref="title"></h1>
      <School></School>
   </div>
</template>
```

通过`this.$ref.title`获取元素，如果是原始HTML元素为DOM，如果是组件标签则获取的是VC。

props属性

向组件中传入参数信息，但是props优先级要比data中的高，但是props属性不应该进行修改。

School.vue

```
<template>
   <div>
      <h3>{{ msg }}</h3>
      <p>{{name}}</p>
      <p>{{year}}</p>
   </div>
</template>

<script>
export default {
   name: 'School',
   data() {
      return{
         msg: 'School info',
      }
  },
   props: ['name', 'year']
}
</script>
```

App.vue

> 如果year变量是Number类型，需要使用`v-bind`进行标识，否则无法识别。

```
<template>
   <div>
      <h1 ref="title"></h1>
      <School name="JNU" :year="1998"/>
   </div>
</template>

.....
```

props类型限制写法：

```
props: {
   name: String,
   year: Number
}
```

props详细写法，可以指定是否必需和默认值：

```
props: {
   name: {
      type: String,
      required: true
  },
   year: {
      type: Number,
      default: 200
  }
}
```

混合mixin

增强代码的复用性

mixin.js

> 相同的数据以原文件为主，不同的数据进行合并

```
export const utils = {
      data(){
            return {x:666}
     },
      methods: {
            hello(){ alert("hello") }
     }
}
```

局部引入：

```
<template>
   <div>
      <h3 @click="hello">{{ msg }}</h3>
   </div>
</template>

<script>
import {utils} from '@/mixin'
export default {
```

```
    name: 'School',
    data() {
        return{
            msg: 'School info',
        }
    },
    mixins: [utils]
}
</script>
```

全局引入的话需要在main.js中进行配置，在其他文件中就无需import和配置mixins属性：

```
import Vue from "vue";
import App from "./App";
import {utils} from '@/mixin'

Vue.mixin(utils)

new Vue({
      el: '#app',
      render: h=>h(App)
})
```

插件

在vue加载的时候可以直接调用，可以用来加载过滤器，自定义指令，mixins以及给原型加方法。

plugins.js

```
export default {
      install(Vue){
            console.log(666);
            // filter directives mixins
            // Vue.filters()
      }
}
```

使用插件：

```
import Vue from "vue";
import App from "./App";
import plugins from "@/plugins";

Vue.use(plugins)

new Vue({
      el: '#app',
      render: h=>h(App)
})
```

自定义事件

父元素：

```
第一种写法
<Dog @ok="demo"></Dog>
<script>
// ...
```

```
    demo(args){
            console.log("Emit ok:"+args)
     }
// ...

</script>

第二种写法
<Cat ref="cat"></Cat>
<script>
// ...
    mounted(){
            this.$refs.cat.$on("ok", method);
            this.$refs.cat.$once("ok", method); // 只触发一次
            this.$refs.cat.$off("ok");  // 解绑一个自定义事件
            this.$refs.cat.$off(["ok","ok2"]);  // 解绑多个自定义事件
     }
// ...

</script>
```

Dog.vue

```
// 触发ok事件
this.$emit("ok", "args")
```

全局事件总线

> 可以实现任意组件间的通信，原理是 `Vue.prototype === VueComponent.__proto__`

首先创建一个全局的通信件：通俗名称为 `$bus`

```
new Vue({
   render: h => h(App),
   beforeCreate() {   // 设为通信件
      Vue.prototype.$bus = this
  }
}).$mount('#app')
```

组件注册事件：

```
mounted() {
   this.$bus.$on('StudentReceiver', (data)=>{
      console.log("Student Recv data", data)
  })
}
```

触发事件：

```
methods:{
   send(){
      this.$bus.$emit('StudentReceiver', '牛逼')
  }
}
```

消息订阅与发布

需要使用到 `pubsub.js` 库

安装： `npm i pubsub-js`

消息订阅与取消：

```
import pubsub from 'pubsub-js'
const pid = pubsub.subscribe('hello', (msgName, msgData)=>{
      // ...
})
pubsub.unsubscribe(pid)
```

消息发送：

```
import pubsub from 'pubsub-js'
pubsub.publish('hello', data);
```

$nextTick使用

一般使用nextTick操作DOM元素

Vue一般都是先进行数据更新-> 模板更新。如果在模板更新之前操作DOM元素操作不会生效，因此需要使用 `vc.$nextTick(()=>{})` 来进行等模板更新之后再操作DOM元素。

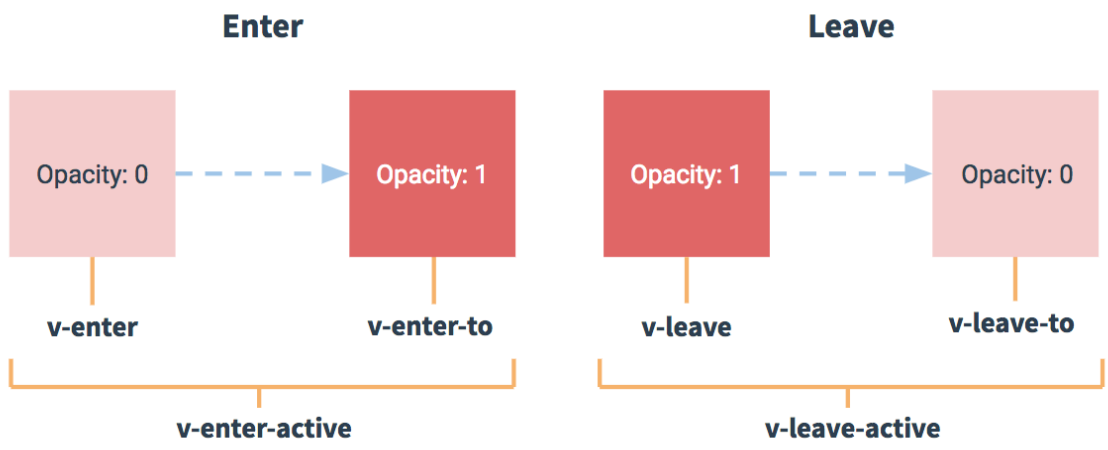
动画与过渡

具体参考 **Vue transitions**

```
<transition name="hello">
      <h1 v-show="isShow"></h1>
</transition>

<style>
      .hello-enter-active{/*....*/}
      .hello-leave-active{/*....*/}
</style>
```

如果不指定 `name` 则默认名称为 `v-xxx-active`



使用第三方库：

```
<link href="https://cdn.jsdelivr.net/npm/animate.css@3.5.1" rel="stylesheet" type="text/css">

<div id="example-3">
   <button @click="show = !show">
     Toggle render
   </button>
   <transition
      name="custom-classes-transition"
      enter-active-class="animated tada"
      leave-active-class="animated bounceOutRight"
   >
      <p v-if="show">hello</p>
   </transition>
</div>
```

插槽slot

🔘默认插槽：

```
<Dog>
    <img src="xxx.jpg"/>
</Dog>
```

Dog.vue

```
<template>
   <div>
      <h2>{{msg}}</h2>
       <slot>这是默认值，无插槽的时候会显示</slot>
      <button @click="send">Send to Student</button>
   </div>
</template>
```

🔘具名插槽：

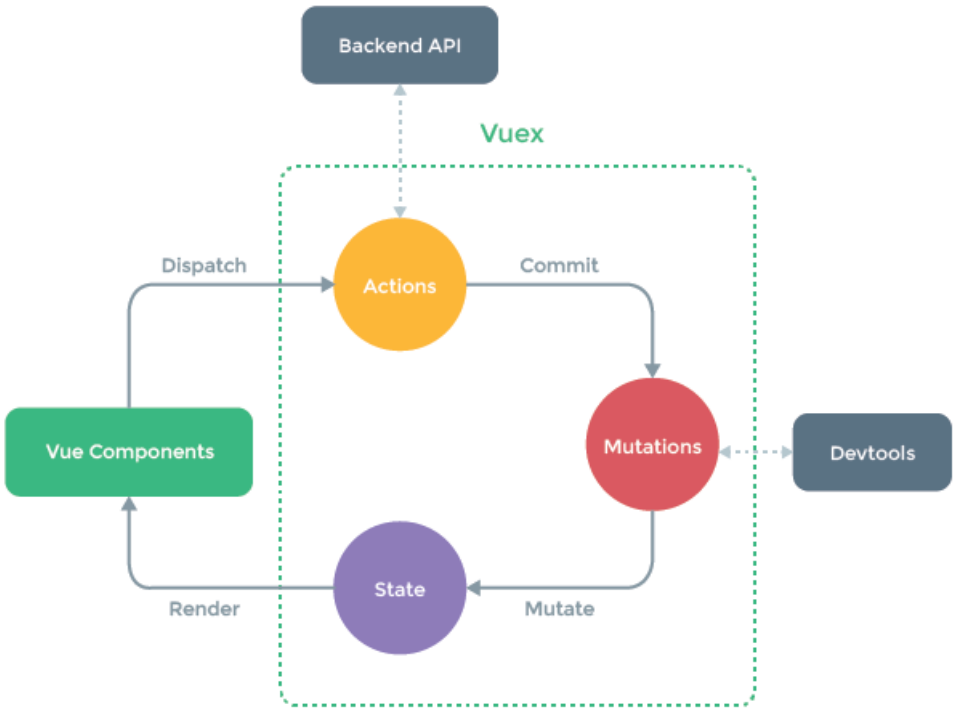
```
<Dog>
    <template v-slot:a_slot><div><img src="xxx.jpg"/></div></template>
    <template v-slot:b_slot><div><img src="xxx.jpg"/></div></template>
</Dog>
```

Dog.vue

```
<template>
   <div>
       <slot name='a_slot'>坑位1</slot>
       <slot name='b_slot'>坑位2</slot>
   </div>
</template>
```

Vuex

vue集中式状态数据管理插件，也是组件间通信的发送，适用于任意组件。



初步准备

安装：`npm i vuex`

工程目录下创建：`store/index.js`文件。

```
import Vue from 'vue'
import Vuex from 'vuex'
const actions = {
      add(context, value){
            console.log('Actions Request Backend Data...')
            context.commit('ADD', value)    // Context commit的事件要一般大写
      }
}
const mutations = {
      ADD(state, value){
            console.log("Mutation recv Actions Data..")
            state.sum += value
      }
}
const state = {
      sum: 0
}
Vue.use(Vuex)
export default new Vuex.Store({
      actions, mutations, state
})
```

需要在index.js中提前使用插件`Vue.use(vuex)`

在Vue实例中添加store字段

```
const vm = new Vue({
   render: h => h(App),
   store,
}).$mount('#app')
```

使用vuex的store，用`dispatch`函数来进行分发行为

```
<template>
   <div>
      <p>{{$store.state.sum}}</p>
      <button @click="add">Send to Student</button>
   </div>
</template>

<script>
export default {
   name: "School",
   methods:{
      add(){
         this.$store.dispatch('add', 1)
      }
  }
}
</script>
```

> 如果在无需请求后端的情况下，可以直接从vc到mutation。`this.$store.commit('ADD', 1)`

getters配置

类似computed属性

```
const state = {
      sum: 0
}

const getters = {
      aSum(state) {
            return state.sum * 10 - 1
      }
}
Vue.use(Vuex)
export default new Vuex.Store({
      actions, mutations, state, getters
})
```

MapState和MapGetters

在模板中通常要使用`this.$store.state.xxx`或者`this.$store.getters.xxx`，前缀很长。

解决方法：

引入：`import {mapState, mapGetters} from 'vuex'`

```
<template>
   <div>
      <h2>{{msg}}</h2>
```

```
      <p>{{sum}}</p>
      <p>{{aSum}}</p>
      <button @click="add">Send to Student</button>
   </div>
</template>

<script>
import {mapState, mapGetters} from 'vuex'
export default {
   name: "School",
   data(){
      return{
         msg: 'School msg'
      }
  },
   methods:{
      add(){
         this.$store.dispatch('add', 1)
      }
  },
   computed:{
      ...mapState({'sum':'sum'}), // 对象写法
      // 可以简写为
      ...mapState(['sum']),       // 数组写法
      ...mapGetters({'aSum':'aSum'})
  }
}
</script>
```

MapActions和MapMutations

同上在代码中通常要使用`this.$store.commit`或者`this.$store.dispatch`，前缀很长。

引入：`import {mapActions, mapMutations} from 'vuex'`

```
methods:{
      ...mapMutations({'sum':'sum'}), // 对象写法,m默认对应生成的函数第一个参数为用户传入的值
      ...mapMutations(['sum', 'sub']),
}
```

类似生成的sum函数：

```
function sum(value){
      // ...
}
```

需要用户在模板上进行自动加参数：

```
<button @click="sum(12)">OK</button>
```

vuex模块化开发

```
const orderOptions = {
      namespaced:true,
      actions:{},
      mutations:{},
      state:{},
```

```
        getters:{}
}

const userOptions = {
        namespaced:true,
        actions:{},
        mutations:{},
        state:{
                age: 20,
                name:'Johb'
        },
        getters:{}
}

export default new Vuex.Store({
        modules:{
                user: userOptions,
                order: orderOptions
        }
})
```

不是用mapState：

```
this.$store.commit('user/Add', params)
```

对于mapState：

```
...mapState(['user', 'order']),     // 数组写法
//或者是
...mapState('user', ['name', 'age']) // 可以直接使用name，不用user.name
```

Vue-router路由

初步

安装：`npm i vue-router`

创建router专用文件夹`router/index.js`，一般将路由组件`xxx.vue`放到`pages`文件夹中

```
import VueRouter from 'vue-router'
import Home from "@/components/Home";
import About from "@/components/About";

export default new VueRouter({
        routes:[
                {path: '/home', component: Home},
                {path: '/about', component: About},
        ]
})
```

使用route：

```
import Vue from 'vue'
import App from './App.vue'

import VueRouter from 'vue-router'
import router from './router'

Vue.config.productionTip = false
Vue.use(VueRouter)

new Vue({
      render: h => h(App),
      router:router
}).$mount('#app')
```

在对应的页面展示中使用 `router-link` 来实现路由的跳转，使用 `router-view` 来展示路由件的位置。

```
<router-link class="list-group-item" active-class="active" to="/about">About</router-link>
<router-link class="list-group-item" active-class="active" to="/home">Home</router-link>

<router-view/>
```

`router-link` 标签的 `replace` 属性设置可以阻止浏览器的后退行为，替换当前的link。

嵌套路由

```
import VueRouter from 'vue-router'
import Home from "@/components/Home";
import About from "@/components/About";
import News from "@/components/News";
import Message from "@/components/Message";

export default new VueRouter({
      routes: [
           {
                  path: '/home',
                  component: Home,
                  children: [
                        {
                              path: 'news',
                              component: News
                        },
                        {
                              path: 'message',
                              component: Message
                        }
                  ]
           },
           {path: '/about', component: About},
     ]
})
```

html

```
<router-link class="list-group-item" active-class="active" to="/home/news">news</router-link>

<router-view/>
```

路由传参

类似于GET请求的参数解析

html:

```
<router-link :to="`/details?id=${m.id}&title=${m.msg}`">{{m.title}}</router-link>
```

解析:

```
<ul>
      <li>Code: {{$route.query.id}}</li>
      <li>Title:{{$route.query.title}} </li>
</ul>
```

命名路由

```
export default new VueRouter({
      routes: [
            {
                  path: '/home',
                  component: Home,
                  children: [
                        {
                              path: 'news',
                              component: News
                        },
                        {
                              path: 'message',
                              component: Message,
                              children: [
                                    {
                                          name: 'msgDetail',
                                          path: 'details',
                                          component: Details
                                    }
                              ]
                        }
                  ]
            },
            {path: '/about', component: About},
      ]
})
```

比如这里的details路由，一般的 `to` 需要写 `/home/message/details` ，添加name属性之后，直接可以使用

```
<router-link :to="{name:'msgDetail'}">{{m.title}}</router-link>
```

RESTful风格($route.params参数)

使用 `/:id` 的类似形式获取

```
export default new VueRouter({
      routes: [
           {
                  path: '/home/:id',
                  component: Home,
           },
     ]
})
```

获取参数:

```
this.$route.params.id
```

编程式路由跳转

使用js控制路由跳转

```
goA(){
       this.$router.push({
             path: '/home'
      })
},
goH(){
    this.$router.replace('/about')
},
this.$router.back()     // 前进
this.$router.forward()  // 后退
```

缓存路由

在进行路由切换的时候会导致原先组件中的数据清除，因此需要缓存路由。使用 `<keep-alive>` 标签让view中的组件保持存活。

```
<keep-alive>
   <router-view/>
</keep-alive>
```

也可以使用 `include` 填写**组件名**，让某个组件一直保持挂载

```
单个
<keep-alive include="news"></keep-alive>
多个
<keep-alive :include="['news','Msg']"></keep-alive>
```

路由组件的生命钩子

两个函数：激活时调用 `activated`，销毁后调用 `deactivated`

```
export default {
   name: "Home",
   activated() {
      console.log('Activated')
  },
   deactivated() {
      console.log("died")
  }
}
```

另外两个，通过路由规则进入的时候调用，也相当于**守卫**：

```
<script>
export default {
   name: "About",
   beforeRouteEnter(to, from, next){
      console.log(to, from)
      // next()
      next()
  },
   beforeRouteLeave(to, from, next){
      console.log('leave')
      next()
  }
}
</script>
```

路由守卫（权限）

一些路由和链接必须满足某种条件才可以进入。

🔘全局前置路由守卫：

```
router.beforeEach((to, from, next)=>{
      console.log(to, from)    // to:  目标路由；form：来源路由
      // 其他逻辑
      next()                     //放行
})
```

如果鉴权信息需要配置可以放在路由信息的 `meta` 属性中：

```
routes: [
     {
            path: '/home',
            component: Home,
            meta: {isAuth: true}
     }]
```

🔘全局后置路由守卫：

用的比较少，可以用于修改网页标题

```
router.afterEach((to, from)=>{
      console.log("latter",to ,from)
})
```

🔘独享守卫前置（无后置）

beforeEnter

```
routes: [
     {
            path: '/home',
            component: Home,
            meta: {isAuth: true},
            beforeEnter: (to, from, next)=>{
                  console.log('Home Enter')
                  next()
           },
           }]
```

UI组件库

移动端：Vant，Cube UI，Mint UI

PC端：element UI，IView UI，Antd