VUE知识点

MADE BY JUNXING YANG

VUE项目结构

el属性

el属性可用于把当前vue实例挂载至某一dom元素上，通过创建el属性，可在由new Vue()生成的vue对象内生成一个$el属性，而$el的值则为当前vue对象挂载至的dom元素。

data对象

存储用于差值语法，逻辑计算等的属性，内部属性通过new Vue最终会在vue对象内被生成，在项目中若想调用这些属性应该通过 'this.' 的方式, this指向当前vue对象，除了可以通过'this.xxx'的方式调用外，还可通过'this.$data.xxx'进行调用(一般不这么用)。

template属性

emplate会根据el所挂载的地方，覆盖默认的html模板，生成由template创建的模版

computed属性

computed用于定义派生属性(计算苏醒)，当data内的某个属性需要输出的是进行计算后的值，再挂载至模版时，可以在computed内进行定义，派生属性必须为一个函数，并且最终在模版内输出的值为其return的结果，由于它依赖于其他属性，所以当其它属性发生变化时，它的输出结果也会发生变化

派生属性的取和写

派生属性同样可以通过 'this.xxx' 的形式进行读取，它不会返回函数体，而是直接返回函数的返回值。
事实上对派生属性进行重写是没有意义的，既然派生属性的依据是data内的属性，那么直接改变data的值就可以了，但是假如真的想直接修改派生属性的话也可以通过把派生属性定义为一个对象然后在对象内分别声明get和set方法,get用于返回派生属性的值，然后set接收一个参数用于对该派生属性所依赖的属性进行赋值。从而改变其自己。

watch属性

watch可以监听数据的变化，并且当某一个属性发生变化时，监听该属性的函数会执行，且会接收两个参数，第一个参数为属性改变后的值，第二个参数为属性改变前的值，监听某一属性的函数应该以该属性的属性名进行命名。

VUE指令

v-text

v-text="data" , v-text实际上与差值语法{{data}}的功能相同，它可以对虚拟dom上的文本内容进行挂载

v-html

v-html可动态生成结构，但尽量少用，因为容易被恶意脚本利用

v-once

只进行初次渲染，即使数据发生改变也不进行二次渲染

v-on

在原生js中，事件回调是无法接受自定义参数的，但在vue中事件回调可以接受参数，我们可以通过 @type="cb('yang',$event)" 进行对事件回调的传参，在vue中$event是事件对象的指定属性名，同时$event的传参位置可以随意设定。

事件修饰符

@click.left => 鼠标左键
@click.right => 鼠标右键
@type.prevent => 取消默认事件
@type.once => 该事件只触发一次
@type.stop => 阻止事件冒泡
@keyup.enter => esc,space,up(方向键)等的按键修饰符

v-bind

用于对虚拟dom上的标签属性进行动态操作的指令

:class

可以动态操作元素的类名，同时还以以一个对象的形式集中操作多个class名，该对象内的键为class类名，属性值应该为布尔值，true代表有该类名，false代表不具备该类名，通过布尔值的切换可实现元素样式状态的切换。同时:class的值还可接受一个数组，用于对多个类名的集中操作。
:class="{red:true,'size-18':false}" ; :class=["red","size-18"]

:style

:style可动态绑定行间属性。
:style="styles"
:style = "[styles,{fontWeight:600}]"
(styles为data内的属性)

:type 可动态操作type属性

v-show

当v-show的值为true时，元素展示，相反，元素不展示。
v-show通过行间属性display:none使元素不展示，
而当v-if的值为false时，该dom元素将压根不会被渲染至页面。

v-if v-else v-else-if

其基本原理与js中的if语句相同，当v-if的值为true时，绑定有v-if的元素会渲染至页面，否则展示绑定了v-else的元素。

v-for

v-for会对目标数组进行循环,然后生成相应的dom元素，其表达式与for in 相同，在遍历数组时，可接收两个变量，1st为值，2st为索引，在遍历对象时，可接收三个变量, 1st为值，2st为键，3st为索引，可以通过 v-for="num in 100"的形式生成100个元素，其次 v-for="str in 'abcd'", 'abcd' 将会发生隐式类型转换。

模板复用

在v-for中存在模板复用，当数据项的顺序发生改变时, vue不会对数据项的顺序进行重排，而只是重排数据项内的内容。
为此可以为数据项添加一个独立的key值，作为vue识别他们的身份标示，当通过 :key 绑定key值时，key值可以为某一数据但key值只可以是原始值，但不能以数组的索引属性作为key值，因为索引是针对位置的索引，而不是值的特征。v-if与v-esle中同样存在模板复用。

数组变异方法

vue对原声数组方法 push , pop , shift , unshift . splice , sort reverse 进行了改写， 当调用以上方法时，将生成一个新的数组替换之前的数组，对于在页面中使用的引用值而言，假如仅仅只是引用值内的数据发生了变化的话，vue是不会对页面进行更新的必须把引用值本身置换掉，vue才会对页面进行更新。
假如想改变引用值内的某个值，而又能促使vue对页面进行刷新的话，可以通过this.$set或Vue.set方法。
this.$set(this.listObj , oldItem , newItem)

v-model

v-model只适用于具有文本输入的元素，如input框。
首先v-model具有:value的功能，在:value指令中，只可以通过js操作改变input框内的文本内容，js操作为起点，input框输出作为终点。
而在v-model中，不仅可以js操作作为起点，input框输出作为终点，还可以通过input框输入作为起点，js内数据被修改作为终点

组件component

全局组件

Vue.component('myCom',{...})

'myCom'为组件名，{}为相关配置项, 若想使用组件，必须在全局进行实例化，即创建一个vue实例(具有el属性)，并在vue环境下才能进行使用。

局部组件

局部组件需在vue实例的components对象内进行使用。

组件中的data属性

组件中的data必须要以一个函数的形式进行声明的原因是，由于组件的复用性，决定了以对象进行定义的方法会导致在不同区域挂载的该组件共用一个data对象，其中一个组件的data属性内的数据发生发生改变的时，便会引起全体组件的数据改变这与闭包中的公有变量同一原理。

而通过函数返回一个对象的方式，可保证该组件在每一次挂载时，data函数所返回对象虽然数据值相同，但相互独立。

组件嵌套

组件中可以再引入组件，全局组件可以在任何一个组件中被引入使用，而组件内定义的局部组件，只能使用于当前组件。

数据传递

父传子

通过在组件标签上挂载 v-bind:xxx = "xxx" , 然后通过组件配置项props属性，以一个数组的方式接收xxx, 即可以接收到父级传来的数据，并且使用。

props

props还可以以对象接收来自父级的数据，如 xxx:{ type:String; required:true; default:'true'}

type规定传过来的值应该是什么类型的值,当然即使所传的值不服合type的要求，该值也可以接收得到，只不过浏览器会弹出注意信息。required规定，当没有进行传值时，在浏览器控制台中是否产生注意信息，default定义数据的默认值。

传值数据的重写

虽然由直接父级传至组件上的原始值，在组件上进行修改并不会影响到父级中的原始值，但vue依然不建议这么做，更好的方法是在组件中应该先把传递来的数据另外存储于一个属性中，然后使用该属性，修改该属性。

假如从直接父级中传过来的是一个引用值，那么当组件上对该引用值进行重写后，在父级中的该引用值也会被重写

子传父

通过在组件标签上定义一个自定义事件如@tellme = "tellme", 并把该事件的回调定义于父级的methods内，在子组件内执行this.$emit("tellme",data),即可通过子级往父级的回调内传值

多值传输

当父级想往子组件传多个值时，:xxx="xxx"的方式会稍显臃余，因此可以把这些需要传递的数据封装至一个对象内，然后通过v-bind="dataObj"的形式传至子组件内，但在组件内依然需要单个单个传值的形式进行接收

动态组件

component标签结合:is属性可以实现动态渲染指定组件,通过把组件名赋值于data对象内的一个属性上，然后再在:is上绑定该属性，该属性所指代的组件将会被渲染至页面，然后通过动态改变该属性的值即可自由切换不同的组件

keep-alive

通过:is动态渲染的input框，当组件发生切换后，之前进行的文本输入将被清空，状态无法被保存会发现，而当为组件包裹一个keep-alive标签作为父级，可以把元素的状态缓存下来

生命周期

步骤

1. new Vue()
2. 事件初识化，声明周期初始化
3. befroCreate()
4. creating: 往vue实例中注入methods,data,props内属性,包括监听器watcher与实例产生关联
5. created()
6. 判断是否有el属性，no：执行$mount(el)方法(组件的情况)，然后无论是否有el属性都会进入实例中是否有template属性的判断， 有的话编译template模版，没有的话编译传统模版。
7. beforeMount()
8. mounting: $el 属性生成，$el覆盖el并锁定相应的dom结构, 此时只是进行了锁定，但还没进行挂载(isMounted:false)。
9. mounted() =>
   替换掉原有dom结构，从新生成一份相同的dom结构，并把vue实例挂载至该新搭建的模版上，isMounted:true。

beforeUdate() updated()

当数据发生变化,并且该数据是应用于页面的数据时，会先触发beforeUpdate()，再触发updated() 需要注意的是这里的updata函数触发的基准不是数据的updata而是dom元素的updata时。

$destory()

vue实例可通过调用原型上的$destory()方法可与其所绑定的dom结构进行解绑，此时再改变进行了数据绑定的属性时， 页面并不会更新，但实例里的数据依然会发生改变。此时beforeDestroy()与destroyed()将先后被执行。

$mount('#app')

当vue实例调用原型上的$mount()方法时可使vue实例重新挂载至指定的dom元素上。

vm._isBeingDestroyed

该属性表示当前的vue实例vm是否曾经被解绑过，当vue实例曾经被解绑过时其值为true，相反为false，当一个vue实例在被解绑状态下从新调用$mount进行了绑定时，再执行$destroy将无法对vue实例进行解绑，也就是当_isBeingDestroyed为true的话，无法进行解绑操作，但该属性不仅可读还可写，所以可以手动把其值改变为false，便可重新进行vue实例的解绑了。

slot插槽

slot插槽用于，当我们希望直接在组件标签内对组件进行改写时，slot插槽可以使组件的灵活性增强实现万能组件以及组件的封装，slot标签就如同函数的形参，而在组件标签内的内容如同实参，组件标签内内容的不同使组件最终所呈现的使用效果也会不同。

solt使用

slot标签应该安置于组件的template模版类，而我们在组件标签里另外输入的内容就像插头一样，插座的位置在哪里，插头就会被插至哪里，也就是说slot标签所在的位置即插头所在的位置。

具名插槽

简单来说具名插槽是一个有专属名字的插槽，可以理解为这个插槽只允许某个或某些人来插，而没有被允许的其他人不能使用该插槽。通过在slot标签上添加上name属性，并定义name值，该slot标签就成了一个具名插槽，而通过在组件标签内的标签上定义slot属性，并使其值与name属性值相同，该标签将会被安插至该插槽上，同样最终dom元素所在的位置，不由位于组件标签内标签的摆放顺序决定，而由template属性内的slot标签的摆放位置决定。而没有声明slot属性的标签将被统一安放至公共插槽上。具名插槽弥补了公共只能对数据进行集中操作无法个别操作的缺陷。假如在组件标签内的其中一个标签已经定义了slot属性，但在template模版中没有相对应的具名插槽的话，即使有公共插槽，该定义了slot属性的标签依然不会被渲染至页面。

而假如在template模版内有未被使用的插槽那么假如该插槽夹带着文本的话，文本将会被渲染至页面。

作用域插槽

只有template模版内的dom结构可以直接使用组件内定义的属性，由于slot标签本身不进行dom结构渲染，所以它即使使用了组件内的属性，并且通过差值语法使用了该属性依然是不会渲染至页面的，而在组件标签内的标签是无法直接使用组件内的属性的，它必须通过组件给它传值后它才能使用组件内的属性，那么假如想实现组件往插头传值的话，必须通过以下步骤：

首先必须在插槽上定义 :data = "data", 然后在插头标签中通过 slot-scope="xxx" 进行接收，slot-scope与props原理相同，都是用于接收传递过来的数据，只有这样，在插头标签中才能使用由组件传过来的数据。

自定义指令

```
<div class="red" v-focus:flag="flag"></div>

Vue.directive('focus', {
    bind() {  //当指令绑定至元素上时触发 },
    inserted() { //当被绑定的元素插入到 DOM 中时触发 },
    update() {  //数据发生变化时触发 },
    componentUpdated() {  //组件发生变化后出触发 },
    unbind() {  //指令与元素解绑时调用 }
})
```

过渡动画

transition的mode属性可规定多个组件下的展示顺序，out-in为正在展示的组件先消失，下一刻要展示的组件再进来，in-out为下一刻要展示的组件先进入，然后正在展示的组件再消失。

```
<transition mode="out-in" name="msg">
    <user-name v-if="flag"></user-name>
    <pass-world v-else></pass-world>
</transition>
```

vue-router

引用

```
<script src="vue-router.js"></script>(非项目)
```

模块化工程

```
npm install vue-router
```

如果在一个模块化工程中使用它，必须要通过 Vue.use() 明确地安装路由功能：在你的文件夹下的 src 文件夹下的 main.js 文件内写入以下代码

```
import VueRouter from 'vue-router'
Vue.use(VueRouter)
```

路由设置方式

```
// 定义路由与绑定组件
var routes = [ { path:"/one",  component:{template:"#a"} },];
// 定义路由
var router = new VueRouter({ routes });
// 创建和挂载实例
new Vue({ el:"#box", router });
```

路由使用方式

```
<div id="box">
    <router-link to="/one">One</router-link>
    <router-view></router-view>
</div>
```

router-link 默认会被渲染成一个a标签，to=""所通向的组件

当该a标签被点击后，浏览器地址栏将会产生"..../one"标识

而路由匹配到的组件，将会渲染在< router-view >

嵌套路由

当一个路由内还需嵌套另一个路由时，可以在原路由配置项上添加一个children数组，进行子路由的配置，并且还需要原组件的dom结构内添加< router-view >来渲染 chlidren 里面的路由视图。

动态路由匹配

```
{ path:"/two:id", component:{template:"#b"} }
```

命名路由

简而言之就是在router-link上不使用to="/xxx"的方式,而是在路由配置项里另外创建一个属性name="xxx",而在路由标签上使用 :to="{ name: 'one'}"，的方式

命名视图

命名视图实质上就是通过一个特定的name，自由配置组件在路由内的位置，如下。

```
<router-view name="a"></router-view>
<router-view name="b"></router-view>
var Foo = { template: '<div>foo</div>' }
var Bar = { template: '<div>bar</div>' }
var routes = [ { path:"/one", components:{  a:Foo,  b:Bar } } ]
```

重定向和别名

通过router内的redirect属性可把路由重定向至另一个组件

```
path: '/a', redirect: '/b'
```

通过router内的alias属性可为路由匹配一个别名，当访问别名时同样会被引导至path所指定的路由，但地址栏上将显示别名

混入

var myMixin = {data:{},methods:{},created:{}.....}
var myCom = {....., mixins: [myMixin] }
简单来说混入对象可以想象成是一个工具，该工具可以安插至任何一个组件里，使该混入对象所拥有的方法，数据成为组件的方法与数据。

混入中的数据合并与覆盖

当组件上data对象与混入对象里的data对象在属性上有冲突时，组件内的data数据将覆盖混入对象的data数据。
同名钩子函数产生冲突时，将会被混为一个数组，组件对象和混入对象内的钩子都会被执行，但混入对象的钩子将先被执行。
值为对象的选项，如methods,components，将被混为同一个对象，两个对象键名冲突时，取组件对象的键值对。

全局混入Vue.mixin({})

可以全局注册混入独享。但应谨慎使用!!全局混入对象将作用于之后所创建的所有Vue实例(包括第三方模版)

$与_

VUE不赞成自主创建变量名以$或_开头的变量，vue担心开发者所设置的开头为$或_的属性会覆盖掉vue内的默认方法或属性，但是开头为$或_的变量名是可以作为vue上的属性被设置上的，只是它不具备数据绑定的功能。
其次在vue实例中每一个以 _ 开头的变量都是该vue实例的私有变量。

$refs对象

可以通过$refs.xxx获取得到在虚拟dom上标记了ref属性的dom元素，设有 v-if并且值为false的元素无法获取。
需要注意的是假如在虚拟dom上的文本区域进行了数据绑定，即使我们通过$refs获取dom元素后，再通过innerText改变元素内的文本，通过数据绑定的文本依然存在。

数据过滤 filters

通过filtersn内定义的函数，可对数据进行过滤，函数的第一个参数为需要被过滤的数据，第二个参数起为自定义参数，通过在虚拟dom上进行{{ 数据名 | 过滤函数(自定义参数) }}的形式便可对数据进行过滤，过滤函数的返回值应该为过滤后的数据。

VUEX

VUEX搭建

1. cnpm install vuex --save
2. 创建store.js文件 src/ store / store.js
3. import Vue from 'vue' ; import Vuex from 'vuex' --store.js
4. Vue.use(Vuex) --store.js
5. export const store = new Vuex.Store({....})
6. import { store } from './store/store' --main.js
7. new Vue({ store:store }) --main.js

基础属性

state: 等价于 data
getters: 等价于 computed
mutations: 等价于 methods (仅限于同步执行的js操作)
actions: 等价于 methods (可接收异步执行的js操作)

state获取

组件想获取store中的state, 可通过computed内注册的计算属性函数体内通过 return this.$store.state.msg 的形式进行获取，然后再在template模板内使用该计算属性即可完成对state中状态的调用。

getters详解及获取

通过 this.$store.getters.msg 即可获取在getters内声明的msg计算属性，当调用该计算属性时，store会往该计算属性内传入两个参数，第一个参数为state, 即store内的state库，第二个参数为getters，即store内的getters库
自定义参数：getters: { msg: (state) => **(id)** => { .... } }

Mutation

更改store中状态的唯一方法是mutation，mutation内声明的函数类似于事件回调，但它不是直接的世界回调，它更像是间接的事件回调。在组件中事件的直接回调中通过执行this.$store.commit('cb' , data) 即可执行在mutation内声明的回调，data为自定义参数，'cb' 的第一个参数为store的state,第二个参数(payload)为自定义参数。在需要传入多个额外参数时，应该往 data 应该为一个对象。
对象提交方式 this.$store.commit ({ type:'cb',amount: data })

Mutation注意事项

假如要给state内的对象添加属性的话应通过
vue.set(obj,'newProp', 123)或直接替换掉原有对象。

Action

Action提交的是mutation,而不是直接变更状态(state), 通过参数context，调用commit触发在mutation内声明的回调，而action的执行则通过在组件中的this.$store.dispatch('actionFn')触发。
action可以接受任何异步操作。

Action与promise、async

既然action可以接受异步任务，那么便可以在Action内的回调中使用promise或async，通过在回调中 return new Promise(...), 以及在组件中执行this.$store.dispatch('actionFn').then(...)
即可通过promise进行异步任务

Module

Vuex允许我们将store分割成模块。每个模块拥有自己的state、mutation、action、getter、甚至是嵌套子模块。

参数的接收(module)

mutation、getters: --》 1st,2st局部 ; 3st,4st全局
1st: state ; 2st: getters ; 3st: rootState 4st: rootGetters
actions: context.state,context.rootState

模块的命名空间

在默认情况下，模块内部的action,mutation,getter是注册在全局命名空间，假如希望模块具有更高的封装度，可以通过添加namespaced : true使其成为带命名空间的模块，在调用时可以通过以下方式, mod为模块名，proterty为具体计算属性名
getters['moduleA/proterty']; dispatch('moduleA/actFn');
commit('moduleA/mutFn')

带命名空间模块访问全局内容

dispatch('gloAction' , null , {root:true})
commit('gloMutation' , null , {root:true})

带命名空间模块注册全局action

```
someAction: {
        root: true,
        handler (namespacedContext, payload) { ... }
}
```

模块动态注册

在store创建之后可以通过store.registerModule动态注册模块
该方法接受两个参数，1st为模块名，2st为相关配置项。
然后可以通过store.state.moduleName的形式访问模块的状态

模块重用

当多个store并且共用同一个模块时，或在一个store中多次注册不同模块名但模块相同的模块时(因为模块本身是对象的问题)，假如在模块中用一个纯对象来声明模块状态的话，那么将会发生模块重用的问题，假如希望多个store中共用的同一模块中的状态是相互独立的话，应该使用函数 return 一个对象，对象内部为状态属性的方式去声明模块的状态

插件(plugins)

vuex的store接受plugins选项(plugins:[myplugin]),该选项暴露出每次mutation的钩子，即plugin本身应该是一个函数，并接收唯一参数store, 通过store.subscribe(cb)的形式，每次mutation被调用时，cb就会执行，并且会传入两个参数分别为mutation与state, mutation为一个对象并且具有两个属性，type指向被调用的mutation, payload为传入被调用的mutation的自定义参数。
通过mutation.type结合if判断，即可针对不同的mutation作出相应的响应或后续操作。同样在插件中不可以直接修改store中的状态，而必须通过提交mutation的方式同步数据源至store

双向绑定与state

假如想通过v-model改变state的话，应该使用带有setter的双向计算属性，在get中return this.$store.state.msg, 在get中提交mutation, this.$store.commit('updateMsg', value)

辅助函数

辅助函数应用于组件提取store数据时，必须在组件中将富足函数从vuex内引入，才可使用

mapState

```
computed: mapState({
    count: state => state.count,  箭头函数式
    countAlias: 'count',    'count'即state.count
    count1 (state) { return state.count + this.localCount }})
 ...mapState([ 'count','count1' ])
```

mapGetters

```
mapGetters({ count: 'count1' })
....mapGetters(['count1'])
```

mapMutations

```
...mapMutations(['count1'])
...mapMutations({ count : 'count1' })
```

mapActions

```
...mapActions(['count1'])
...mapActions({ count : 'count1' })
```

模块中使用辅助函数

```
...mapState('module1', { a: state => state.a })
...mapActions('module1', [ 'count1' ])
```

统一设置路径

```
import { createNamespacedHelpers } from 'vuex'
const { mapState, mapActions, mapGetters } =
createNamespacedHelpers('module1')
```

统一设置路径后，无需在辅助函数内单独设置路径