Vue核心语法

响应式数据与插值表达式

插值表达式使用两个大括号表示，可以是data数据，也可以是表达式，也可以是方法函数

```
<body>
    <div id="app">
        <p>{{ title }}</p>
        <p>{{ content }}</p>
        <p>{{ 1+2+3 }}</p>
        <p>{{ 1>2 ? '对':'错' }}</p>
        <p>{{ output() }}</p>
        <p>{{ outputContent }}</p>
    </div>
</body>
```

可以把下面的vm理解成一个方法类，`data`就是`__init__`，`methods`就是方法类中的其他方法

```
<script>
    const vm = new Vue({
        // el设置生效范围
        el: '#app'
        // data设置数据，可以在控制台直接输入vm.title获取或者修改对应的title值
        data () {
            return {
                title: '这是标题文本',
                content: '这是内容文本',
                htmlContent: '这是一个<span>span</span>标签',
                arr: [ 'a','b','c','d' ],
                obj: { a:10 , b:20 , c:30 },
                bool: true,
                inputValue: "默认内容"
            }
        },
        // methods设置方法，例如下面的output()方法
        methods: {
            output(){
                console.log('method执行了')
                return '标题为: ' + this.title + ',内容为' + this.content
```

```
            }
        },
        // computed计算属性，和methods类似，但是有缓存机制，注意：computed是一个属性而不是方法
        // 缓存机制：如果再次请求的数据不变则不进行二次计算
        computed: {
            outputContent(){
                console.log('method执行了')
                return '标题为：' + this.title + ',内容为' + this.content
            }
        },
        // watch侦听器，侦听对应的属性，如果属性变化则进行对应的操作，例如监听下面的title属性
        watch: {
            title(newValue,oldValue){
                console.log(newValue,oldValue)
            }
        }
    })
</script>
```

指令

v-text

内容指令；直接输出文本，会覆盖掉原始内容：

`<p v-text="htmlContent">123</p>` == `这是一个<span>span</span>标签`

v-html

内容指令；会输出文本，如果是html则会解析成html，会覆盖掉原始内容

`<p v-html="htmlContent">123</p>` == `这是一个span标签`

v-for

渲染指令；可以循环输出可迭代对象，例如列表或者集合

`<p v-for="item in arr">这是内容{{item}}</p>` == `这是内容a\n这是内容b...`

`<p v-for="(item,key,index) in obj">这是内容{{item}}{{key}}{{index}}</p>`

== `这是内容10a0\n这是内容20b1...`

v-if

渲染指令；可以控制一个元素的创建与销毁，如果为`true`则创建，如果为`false`则销毁

销毁效果等于：`element.clean()`

`<p v-if="false">标签内容</p>` ==

`<p v-if="true">标签内容</p>` == `标签内容`

v-show

渲染指令；控制一个元素的显示与隐藏，如果为`true`则显示，如果为`false`则隐藏

隐藏效果等于：`display:none`

`<p v-show="bool">标签内容</p>` ==

`<p v-show="bool">标签内容</p>` == `标签内容`

v-bind

属性指令；在标签内部的内容无法使用插值表达式绑定数据，可以使用v-bind来进行标签内数据绑定

`<p title="这是标题内容">这是外部内容</p>` == 鼠标移到`这是外部内容`上面：`这是标题内容`

`<p v-bind:title="title">这是外部内容</p>` == 鼠标移到`这是外部内容`上面：`这是标题内容`

也可以简写成`:`

`<p :title="title">这是外部内容</p>` == 鼠标移到`这是外部内容`上面：`这是标题内容`

v-on

事件指令；给当前标签绑定一个事件

`<button v-on:click="output">按钮</button>`

也可以简写成`@`

`<button @click="output">按钮</button>`

v-model

表单指令；只能用于表单，v-model可以实现双向数据绑定

`<input type="text" v-model="inputValue"></input>` == `默认内容` -> `默认内容123`

`<p v-text="inputValue"></p>` == `默认内容` -> `默认内容123`

修饰符

用来实现一些简单功能

- `trim`：去除两端空格 `<input type="text" v-model.trim="inputValue"></input>`

Vue脚手架与组件化开发

目录结构

- package.json：包含两个常用命令，serve and build，一个用来运行，一个用来打包
- src：主要代码目录
  - main.js：主要js文件，包含了vue的实例
  - App.vue：启动页面，类似于index.html
  - components：组件，里面包含了所有的组件

components

每一个组件都包含三个部分：

template

等效于一个html，但是不同的是一个 `template` 有且只有一个div，所有内容都在div中

script

脚本，一般在这里实现其他组件的导入

例如：`import HelloWorld from './components/HelloWorld.vue'`

导入的组件都可以在 `template` 中当作一个标签使用

例如：`<HelloWorld msg="Welcome to Your Vue.js App"/>`

注意一点：el属性只能出现在根组件中，普通的组件不需要使用el属性

style

样式

> 每一个vue文件的运行都是一次实例化， `components` 中的vue可以在 `App.vue` 中实例化，而 `App.vue` 可以在 `main.js` 中实例化

组件间的通信

父传子

父传子：通过 `props` 进行处理

`HelloWorld.vue` ，子组件，接收信息

```
<script>
export default {
  name: 'HelloWorld',
  // 设置通信相关
  props: {
    // 通信名称：通信类型
    msg: String,
    // 通信名称：通信信息{类型、默认值、是否必须}
    count: {
        type: [String,Number],
        //default: 100,
        // 该信息是否必须，如果为true则必须，空值报错
        required: true
    }
  }
}
</script>
```

`App.vue` ，父组件，发送信息

```
<template>
  <div id="app">
    <img alt="Vue logo" src="./assets/logo.png">
    <HelloWorld
        msg="Welcome to Your Vue.js App"
        count="10"
    ></HelloWorld>
```

```
  </div>
</template>

<script>
// 导入才能进行上面的实例化，不导入上面就会报错
import HelloWorld from './components/HelloWorld.vue'

export default {
  name: 'App',
  components: {
    HelloWorld
  }
}
</script>
```

子传父

子传父：通过自定义事件来完成

`HelloWorld.vue`，子组件，发送信息

```
<script>
export default {
  name: 'HelloWorld',
  // 创建一个数据并创建一个方法，当这个方法被调用时会改变这个数据
  // 目标是把这个数据发送给父组件
  data() {
      return{
          childCount: 0
      }
  },
  methods: {
      handler(){
          this.childCount++
          // $emit()用于触发自定义事件，当事件触发时就传递一个数据
          // 第一个参数表示触发的事件，第二个参数表示事件要传递的数据
          this.$emit('child-count-change', this.childCount)
      }
  }
}
</script>
```

`App.vue`，父组件，接收信息

```vue
<template>
  <div id="app">
    <img alt="Vue logo" src="./assets/logo.png">
    <HelloWorld
        msg="Welcome to Your Vue.js App"
        count="10"
        @child-count-change="handler"
    ></HelloWorld>
  </div>
</template>

<script>
// 导入才能进行上面的实例化，不导入上面就会报错
import HelloWorld from './components/HelloWorld.vue'

export default {
  name: 'App',
  components: {
    HelloWorld
  },
  data() {
      childData: 0
  },
  methods: {
      // 当上面的函数被触发时，会调用下面这个函数并得到一个对面传来的数据
      // 这个数据可以随意改造，在函数中做任何处理，下面简单赋个值
      handler(childCount) {
          this.childData = childCount
      }
  }
}
</script>
```

整个流程如下：

- 子组件在需要的函数中使用 `$emit` 来自定义个事件，当这个函数被调用时，会触发这个事件
- 这个事件如果在父组件中，父组件可以获取到对应传过来的值，也可以设置一个函数，来处理这个值

同级传递

三种方法

1. 通过父级组件做中转
2. 创建一个专门的中转组件

3. 使用vue官方的全局状态管理工具vuex进行状态的管理

组件插槽

父组件调用子组件的时候，可以直接使用单标签，也可以使用双标签，双标签的话，标签之间就可以放入一些内容，这些内容可以在子标签中详细设置，类似于插槽

插槽有三种：普通插槽，具名插槽，作用域插槽

```
// 父组件
<template>
  <div id="app">
    <HelloWorld>这是普通插槽，可以放到子组件中</HelloWorld>
    <HelloWorld>
        <template v-slot:footer>这是具名插槽，可以使用v-slot来区分</template>
        <template #footer>v-slot可以简写成#</template>
        <template #footer2="{childCount}">作用域插槽，可以接收值{{childCount}}，但只能在当前插槽中接收</template>
    </HelloWorld>
  </div>
</template>

// 子组件
<template>
  <div id="app">
      <h1>这是正常子组件信息</h1>
      <slot>普通插槽，不使用插槽就使用当前的默认文本</slot>
      <slot name="footer">具名插槽，可以使用name来进行区分</slot>
      <slot name="footer2" :childCount="childCount">作用域插槽，可以传入自己的值</slot>
  </div>
</template>
```

VueRouter & Vuex

基本概念

什么是router

route：单个路由，本质上就是一个路径与一个视图的对应关系，可以看成一个名片（姓名--手机号）

routes：由route组成的一组路由，可以看作一个电话簿

router：路由器，用来管理路由，从用户那里拿到路径，然后在routes中找对应的视图并返回，可以看成接线员

vue-router

模式：`vue-router`分两种模式：`history` and `hash`

- `history`：地址栏没有`#`，更美观但不如`hash`实用
- `hash`：默认的，地址栏中带有`#`，这种模式不会跳转，和单页面很搭

组件：`vue-router`增加两个组件：`router-link` and `router-view`

- `<router-link to="/foo">Go to Foo</router-link>`相当于`<a href="/foo">Go to Foo</a>`
- `router-view`起到类似占位符的作用，匹配路径的组件会被显示在此处

```
<template>
  <div id="app">
    <nav>
      <!-- 使用 router-link 组件来导航，`to` 属性指定链接 -->
      <!-- <router-link> 默认会被渲染成一个 `<a>` 标签 -->
      <router-link to="/">Home</router-link> |
      <router-link to="/about">About</router-link>
      <!-- to也可以使用name的方式 -->
      <router-link :to="{name:'about'}">About</router-link>
    </nav>
    <!-- 路由出口 -->
    <!-- 路由匹配到的组件将渲染在这里 -->
    <router-view/>
  </div>
</template>
```

什么是vuex

一个统一的数据存储方式，可以在任何组件中进行访问

目录结构

- components：用来存放组件
- router：用来存放路由
  - index.js：路由表，在`routes`中设置路由对应关系；在`router`的`mode`中设置`history` or `hash`
- store：用来存放vuex相关
  - index.js：全局数据表，或许可以理解成一个UserData.json
- views：用来存放视图
- App.vue：主页或者说是起始页
- main.js：用来导入对应的包，例如vue\app\router\store

router/index.js

```
import Vue from 'vue'
import VueRouter from 'vue-router'
// 导入对应的组件视图，不导入后面无法使用
import HomeView from '../views/HomeView.vue'

Vue.use(VueRouter)

const routes = [
  // 路由表，一个字典对应一个路由，其中path是url路径，name是路径别名，component是对应的组件
  // 换句话说：path是姓名；name是外号；component是手机号
  {
    path: '/',
    name: 'home',
    component: HomeView
  },
  {
    path: '/about',
    name: 'about',
    //路由级别的代码拆分
    //这会为该路由生成一个单独的块（about.[哈希值].js），该路由在访问时会被延迟加载。
    component: () => import(/* webpackChunkName: "about" */ '../views/AboutView.vue')
  }
]

const router = new VueRouter({
  // history就是不带 # 的路径，如果把这行删除或者注释掉就会变成默认的hash模式
  mode: 'history',
  base: process.env.BASE_URL,
```

```
  routes
})

export default router
```

store/index.js

```
import Vue from "vue";
import Vuex from "vuex";

Vue.use(Vuex);

export default new Vuex.Store({
  state() {
    // 设置好属性之后，就可以在任何组件中使用了
    // 使用 this.$store.state.loginStatus进行调用
    return {
      loginStatus: "用户已经登录",
      count: 0,
    };
  },

  // getters可以理解成计算属性，有一个缓存功能
  // 使用 this.$store.getters.len 调用
  getters: {
    len(state) {
      return state.loginStatus.length;
    },
  },

  // 如果想要修改数据，在这里面先封装方法，再修改
  // 使用 this.$store.commit('changeCount',1) 进行调用
  mutations: {
    // 这里可以把state理解成this，就是自己目标对象的意思，
    // 所以只需要传入一个参数num就够了
    changeCount(state, num) {
      state.count += num;
      console.log("mutation执行了，count值:", state.count);
    },
  },

  // actions是用来做异步包装的,把mutations的操作又封装了一层
  // 只要是异步操作，就在这里写
  // 使用this.$store.dispatch('delayChangeCount',3)调用
  actions: {
```

```
    delayChangeCount(store, num) {
      setTimeout(() => {
        store.commit("changeCount", num);
      }, 3000);
    },
  },

  // 类似于路由中的children
  // 使用 this.$store.a.commit('changeCount',1) 调用
  modules: {
    a:{
      state,
      mutations,
      ...
    },
    b:{
      state,
      mutations,
      ...
    },
  },
});
```

路由使用

配置一个路由

分为三步：

1. 在 `views` 创建一个 `view` 视图，例如： `NewView.vue` ，并编写对应的页面代码
2. 在 `router/index.js` 中导入这个视图，
   例如： `import NewView from '../views/NewView.vue'`
3. 在 `router/index.js` 中的 `routes` 中添加对应的字典，
   例如： `{ path: '/new', name: 'new', component: NewView }`

动态路由

动态路由是指可以根据不同的需求动态加载不同的路由，实现不同的页面渲染。

例如由两百个文章，对应的文章id从001到200。文章的主题结构都是一样的，不需要创建200个 `view` ，这就需要使用动态路由了

在 `path` 中使用 `:` 即可实现对应的功能

例如：`{ path: '/new/:id', name: 'new', component: NewView }`

如果希望数据可以在组件内部访问，只需要在 `path` 中添加一个 `props: true` 即可，

例如：`{ path: '/new/:id', name: 'new', component: NewView, props: true}`

对应的 `newView` 中：

```
<script>
    export default {
        name: 'NewView',
        props: ['id']
    }
</script>
```

嵌套路由

就是路径的多级目录，如何设置？例如/new/55/comment这种？使用嵌套路由

```
  {
    path: "/new/:id",
    name: "new",
    component: NewView,
    children: [
        {path: "comment1", name: "comment", component: NewViewComment1},
        {path: "comment2", name: "comment", component: NewViewComment2},
    ],
    props: true,
  },
```

导航

编程式导航

系统自动的导航

`this.$router.push({name:'home'})`

导航守卫

每次导航的时候都需要运行的部分