Vue全家桶 & 原理

复习

组件化

作业

1. 尝试解决Input里面$parent派发事件不够健壮的问题

   element的minxins方法

   Input组件中的使用

2. 组件实例创建的另一种解决方案

```
const Ctor = Vue.extend(Component)
const comp = new Ctor({propsData: props})
comp.$mount();
document.body.appendChild(comp.$el)
comp.remove = () => {
    // 移除dom
    document.body.removeChild(comp.$el)
    // 销毁组件
    comp.$destroy();
}
```

3. 使用插件进一步封装便于使用，create.js

```
import Notice from '@/components/Notice.vue'
//...
export default {
  install(Vue) {
    Vue.prototype.$notice = function (options) {
      return create(Notice, options)
    }
  }
}
```

资源

1. vue-router
2. vuex
3. vue-router源码
4. vuex源码

知识点

vue-router

Vue Router 是 Vue.js 官方的路由管理器。它和 Vue.js 的核心深度集成，让构建单页面应用变得易如反掌。

安装：`vue add router`

核心步骤：

- 步骤一：使用vue-router插件，router.js

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

- 步骤二：创建Router实例，router.js

```
export default new Router({...})
```

- 步骤三：在根组件上添加该实例，main.js

```
import router from './router'
new Vue({
  router,
}).$mount("#app");
```

- 步骤四：添加路由视图，App.vue

```
<router-view></router-view>
```

- 导航

```
<router-link to="/">Home</router-link>
<router-link to="/about">About</router-link>
```

vue-router源码实现

需求分析

- 作为一个插件存在：实现VueRouter类和install方法
- 实现两个全局组件：router-view用于显示匹配组件内容，router-link用于跳转
- 监控url变化：监听hashchange或popstate事件
- 响应最新url：创建一个响应式的属性current，当它改变时获取对应组件并显示

实现一个插件：创建VueRouter类和install方法

创建kvue-router.js

Vue本身是一个构造函数，其使用方式是new Vue(),返回一个Vue实例

```
let Vue; // 引用构造函数，VueRouter中要使用

// 保存选项
class VueRouter {
  constructor(options) {
    this.$options = options;
  }
}

// 插件：实现install方法，注册$router
VueRouter.install = function(_Vue) {
  // 引用构造函数，VueRouter中要使用
  Vue = _Vue;

  Vue.mixin({
    beforeCreate() {
      // 只有根组件拥有router选项
      if (this.$options.router) {
        // vm.$router
        Vue.prototype.$router = this.$options.router;
      }
    }
  });
};

export default VueRouter;
```

不使用import方式添加Vue是因为这个本身是一个插件(install方法)，通过install方式被使用之后，会直接使用到install它的项目的Vue，所以用let就可以，如果用import，则会在打包时把整个Vue都打包，体积很大

用静态方法定义"."

此处传入的_Vue，本质上被install之后传入的也是安装这个插件的项目的Vue，以示区分就使用_Vue

此处可以理解为自己赋值给自己

使用全局混入，beforeCreate()这个钩子函数会在这个项目的所有组件中都执行一次，如果某个组件中有beforeCreate函数则会和该组件的beforeCreate函数融合起来，并不会完全替代该组件的beforeCreate函数

> 为什么要用混入方式写？主要原因是use代码在前，Router实例创建在后，而install逻辑又需要用到该实例

不能使用Vue.component('router-view',{template:``})的方式去写，使用webpack的情况下，最终打包的时候已经无法编译template，所以只能使用render函数

创建router-view和router-link

创建krouter-link.js

```
export default {
  props: {
    to: String,
    required: true
  },
  render(h) {
    // return <a href={'#'+this.to}>{this.$slots.default}</a>;
    return h('a', {
      attrs: {
        href: '#' + this.to
      }
    }, [
      this.$slots.default
    ])
  }
}
```

h(tag,data,children)

渲染后内容 <router-link to="/about">xxxx</router-link> 根据render函数的要求，此处的xxx需要渲染为<router-link>的children元素，所以可以使用slot来替代纯文本

通用写法，可以脱离vue-cli使用

限制只能在vue-cli中的写法：JSX 此时不定义h
{this.$slots.default

创建krouter-view.js

```
export default {
  render(h) {
    // 暂时先不渲染任何内容
    return h(null);
  }
}
```

监控url变化

定义响应式的current属性，监听hashchange事件

hashchange事件是html5新增的api,用来监听浏览器链接的hash值变化

```
class VueRouter {
  constructor(options) {
    // current应该是响应式的
    Vue.util.defineReactive(this, 'current', '/')

    // 定义响应式的属性current
    const initial = window.location.hash.slice(1) || '/'
    Vue.util.defineReactive(this, 'current', initial)

    // 监听hashchange事件
    window.addEventListener('hashchange', this.onHashChange.bind(this))
    window.addEventListener('load', this.onHashChange.bind(this))
  }

  onHashChange() {
    this.current = window.location.hash.slice(1)
  }
}
```

响应式方法

和下面重复，以下面为准

监听路径变化时

监听页面刷新时

动态获取对应组件，krouter-view.js

```
export default {
  render(h) {
    // 动态获取对应组件
    let component = null;
    this.$router.$options.routes.forEach(route => {
      if (route.path === this.$router.current) {
        component = route.component
      }
    });
    return h(component);
  }
}
```

提前处理路由表

提前处理路由表避免每次都循环

```
class VueRouter {
  constructor(options) {
    // 缓存path和route映射关系
    this.routeMap = {}
    this.$options.routes.forEach(route => {
      this.routeMap[route.path] = route
    });
  }
}
```

使用，krouter-view.js

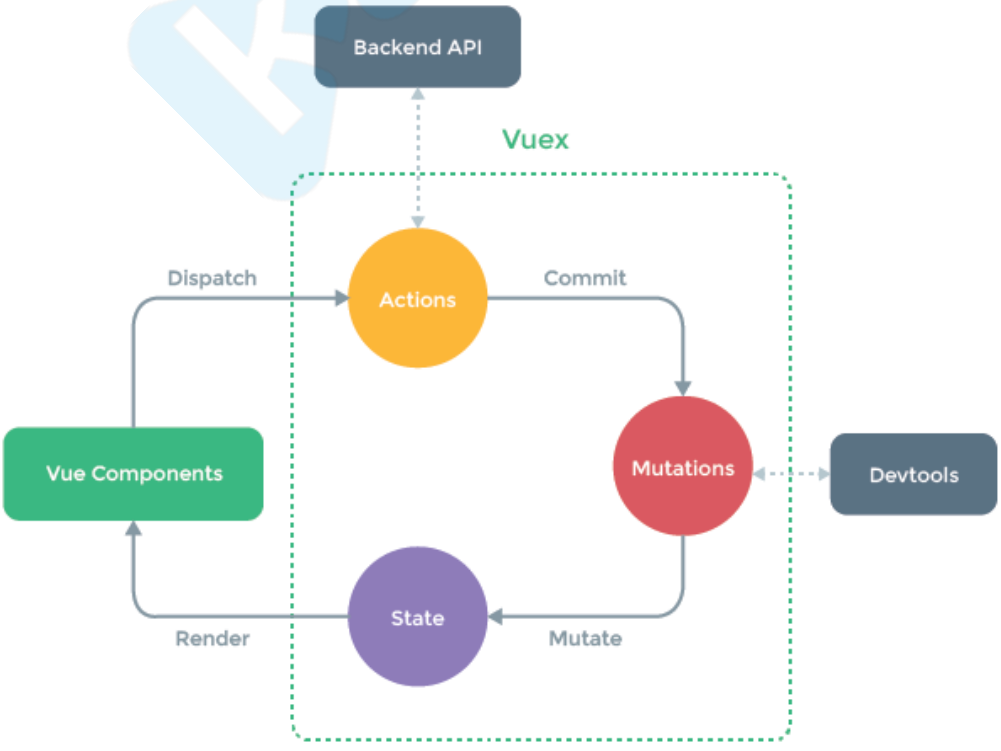
```
export default {
  render(h) {
    const {routeMap, current} = this.$router
    const component = routeMap[current].component;
    return h(component);
  }
}
```

由于使用了自定义的VueRouter，所以this.$router就是自定义的VueRouter，带有routeMap

> 整理一下

Vuex

Vuex **集中式**存储管理应用的所有组件的状态，并以相应的规则保证状态以**可预测**的方式发生变化。



整合vuex

```
vue add vuex
```

核心概念

- state 状态、数据
- mutations 更改状态的函数
- actions 异步操作
- store 包含以上概念的容器

状态 - state

state保存应用状态

```
export default new Vuex.Store({
  state: { counter:0 },
})
```

状态变更 - mutations

mutations用于修改状态，store.js

```
export default new Vuex.Store({
  mutations: {
    add(state) {
      state.counter++
    }
  }
})
```

派生状态 - getters

从state派生出新状态，类似计算属性

```
export default new Vuex.Store({
  getters: {
    doubleCounter(state) { // 计算剩余数量
      return state.counter * 2;
    }
  }
})
```

动作 - actions

添加业务逻辑，类似于controller

```
export default new Vuex.Store({
    actions: {
        add({ commit }) {
            setTimeout(() => {
                commit('add')
            }, 1000);
        }
    }
})
```

此时在add中使用了定时器，所以此时commit中的this会指向window

测试代码：

```
<p @click="$store.commit('add')">counter: {{$store.state.counter}}</p>
<p @click="$store.dispatch('add')">async counter: {{$store.state.counter}}</p>
<p>double: {{$store.getters.doubleCounter}}</p>
```

vuex原理解析

任务分析

- 实现一个插件：声明Store类，挂载$store
- Store具体实现：
  - 创建响应式的state，保存mutations、actions和getters
  - 实现commit根据用户传入type执行对应mutation
  - 实现dispatch根据用户传入type执行对应action，同时传递上下文
  - 实现getters，按照getters定义对state做派生

初始化：Store声明、install实现，kvuex.js：

vuex四种状态：state,getters,mutations,actions
执行mutations用commit(可以理解为同步方法)
执行actions用dispatch(可以理解为异步方法)

```
let Vue;

class Store {
  constructor(options = {}) {
    this._vm = new Vue({
      data: {
        $$state:options.state
      }
    });
  }
  get state() {
    return this._vm._data.$$state
  }
  set state(v) {
    console.error('please use replaceState to reset state');
  }
}

function install(_Vue) {
  Vue = _Vue;
```

此处使用new Vue是为了把传入的state在data中变成一个响应式的状态

使用$$，则vue在创建的时候不会做代理，即vm无法直接访问如此定义的属性，对外部隐藏

存取器只能在类中定义

有了存取器，则用户直接访问的是存取器返回的内容，而非直接访问contructor里面的$$state

存取器

此处的v代表假如用户试图直接修改state时传入的数据

```
  Vue.mixin({
    beforeCreate() {
      if (this.$options.store) {
        Vue.prototype.$store = this.$options.store;
      }
    }
  });
}

export default { Store, install };
```

此处Store类没有把install方法直接当成其静态方法，所以export的时候要同时export install方法

实现commit：根据用户传入type获取并执行对应mutation

```
class Store {
  constructor(options = {}) {
    // 保存用户配置的mutations选项
    this._mutations = options.mutations || {}
  }

  commit(type, payload) {
    // 获取type对应的mutation
    const entry = this._mutations[type]

    if (!entry) {
      console.error(`unknown mutation type: ${type}`);
      return
    }
    // 指定上下文为Store实例
    // 传递state给mutation
    entry(this.state, payload);
  }
}
```

type可以理解成函数名

payload是用户传进来的参数,不包括state

根据例如mutations:{
add(state){
console.log("!!!")
}
}

首先要传入state,再传入其他参数

实现actions：根据用户传入type获取并执行对应mutation

```
class Store {
  constructor(options = {}) {
    // 保存用户编写的actions选项
    this._actions = options.actions || {}

    // 绑定commit上下文否则action中调用commit时可能出问题!!
    // 同时也把action绑了，因为action可以互调
    const store = this
    const {commit, action} = store
    this.commit = function boundCommit(type, payload) {
      return commit.call(store, type, payload)
    }
    this.action = function boundAction(type, payload) {
      return action.call(store, type, payload)
    }
  }

  dispatch(type, payload) {
    // 获取用户编写的type对应的action
    const entry = this._actions[type]
```

此处要用已经保存好的store(this)代替直接写this，如果直接写有可能是指向this.commit/action

```
    if (!entry) {
      console.error(`unknown action type: ${type}`);
      return
    }
    // 异步结果处理常常需要返回Promise
    return entry(this, payload);
  }
}
```

作业

1. 尝试去看看vue-router的源码，并解答：嵌套路由的解决方式
2. 尝试去看看vuex的源码，并实现getters的实现
3. 了解vue数据响应原理为下节课做准备