vue

基本使用:

1） 引入 vue.js

2） 创建 vue 实例对象（vm），指定选项（配置）对象

- el:指定 dom 标签容器的选择器
- data:指定 初始化状态数据的对象/函数(返回一个对象)

3） 在页面模板中使用{{}}或 vue 指令

Data 在组件中使用必须要是一个函数的原因:

实际上，它首先需要注册一个组件构造器，然后注册组件，注册组件的本质其实就是建立一个组件构造器的引用,使用组件才是真正的创建一个组件实例，所以注册组件并不产生新的组件类，但会产生一个可以用来实例化的新方式，看下面的例子

```
var MyComponent = function() {}
MyComponent.prototype.data = {
  a: 1,
  b: 2,
}
// 上面是一个虚拟的组件构造器，真实的组件构造器方法很多
var componentA = new MyComponent()
var componentB = new MyComponent()
// 上面实例化出来两个组件实例，也就是通过<my-component>
```

调用，创建的两个实例

```
componentA.data.a === componentB.data.a // true
componentA.data.b = 5
componentB.data.b // 5
```

这样处理的话两个实例对象引用同一个对象，那么在修改其中一个属性的时候，另外一个实例也会跟着改变。所以应该这样处理

```
var MyComponent = function() {
  this.data = this.data()
}
MyComponent.prototype.data = function() {
  return {
    a: 1,
    b: 2,
  }
}
```

这样每一个实例的 data 属性都是独立的，不会相互影响了，所以 vue 组件的 data 必须是函数是 js 本事特性所带来的，与 vue 设计无关。

Vue 对象的选项

1） el

- 指定 dom 标签容器的选择

- Vue 会管理对应的标签及其子标签

2） Data

- 对象或者函数类型
- 指定初始化状态属性数据的对象
- Vm 也会自动拥有 data 中所有属性
- 页面中可以直接访问使用
- 数据代理：由 vm 来代理对 data 中所有属性的操作

3） Methods

- 包含多个方法的对象
- 供页面中的事件指令来绑定回调
- 回调函数默认有 event 参数，但也可以指定自己的参数
- 所有的方法由 vue 对象来调用，访问 data 中的属性直接使用 this.xxx

4） Computed

- 包含多个方法的对象
- 对状态属性进行计算返回一个新的数据，供页面获取显示
- 一般情况下是相当于一个只读的属性
- 利用 set/get 方法来实现属性数据的计算读取，同时监视属性数据的变化
- 如何给对象定义 get/set 属性：
  - 在创建对象时指定：get name(){return xxx} set

name(value){}

- 在对象创建之后指定：Object.defineProperty(obj,age,{get(){},set(value){}})

5） Watch

- 包含多个属性监视的对象
- 分为一般监视和深度监视
  - Xxx:function(value){}
  - Xxx:{
    Deep:true,
    Handler:fun(value)
    }
- 另一种添加监视方式：vm.$watch(‘xxx’,function(value){})

Vue 内置指令:

v-text/v-html: 指定标签体

*v-text：当作纯文本

*v-html：将 value 作为 html 标签来解析

v-if v-else v-show: 显示/隐藏元素

*v-if：如果 vlaue 为 true, 当前标签会输出在页面中

*v-else：与 v-if 一起使用, 如果 value 为 false, 将当前

标签输出到页面中

*v-show: 就会在标签中添加 display 样式, 如果 vlaue 为 true, display=block, 否则是 none

v-for：遍历

*遍历数组：v-for="(person, index) in persons"

*遍历对象：v-for="value in person" $key

v-on：绑定事件监听

*v-on:事件名, 可以缩写为:@事件名

*监视具体的按键: @keyup.keyCode @keyup.enter

*停止事件的冒泡和阻止事件默认行为: @click.stop @click.prevent

*隐含对象: $event

v-bind：强制绑定解析表达式

*html 标签属性是不支持表达式的, 就可以使用 v-bind

*可以缩写为: :id='name'

*:class

*:class="a"

*:class="{classA：isA, classB：isB}"

*:class="[classA, classB]"

*:style

:style="{color：color}"

v-model

*双向数据绑定

*自动收集用户输入数据

ref：标识某个标签

*ref='xxx'

*读取得到标签对象：this.$refs.xxx

事件传参的时候 默认传入事件名event,多于一个参数的时候调用要用$event 占位

动画:xxx 为 transition 标签设置的 name 值

方法一：单独设置 class 属性

```
xxx-enter
xxx-leave
xxx-enter-active
xxx-leave-active
xxx-enter-to
xxx-leave-to
```

方法二:animation 属性

```
xxx-leave-active{animation:bounce-in .5s reverse;}
@key-frames bounce-in {
   0%{
      transfrom:scale(0);
```

```
    }
    50%{
        transform:scale(1.5);
    }
    100%{
        transform:scale(1);
    }
}
```

定义过滤器:

```
Vue.filter(filterName,function(value[,arg1,arg2...]){
    进行一定的数据处理
    return newValue
})
```

在使用模板{{}}的时候会出现闪现现象,使用 v-clock 防止闪现

v-clock 与 css 配合[v-clock]{display:none}

自定义指令 (全局指令和局部指令)

1.注册全局指令(在 js 全局中注册)

```
Vue.directive('my-directive',function(el,binding){
    el.innerHTML = binding.value.toUpperCase()
```

```
})
```

2.注册局部指令(在 vue 实例里注册)

```
directives:{
    'my-directive':{
        bind(el,binding){
            el.innerHTML = binding.value.toUpperCase()
        }
    }
}
```

3.使用指令

```
v-my-directive='xxx'
```

定义自定义 vue 插件

```
(function () {
    const MyPlugin = {}
    MyPlugin.install = function (Vue, options) {
        1. 添加全局方法或属性
        Vue.myGlobalMethod = function () {
            alert('Vue 函数对象方法执行')
        }
        2. 添加全局资源
        Vue.directive('my-directive',    function    (el,
```

```
binding) {
        el.innerHTML = "MyPlugin my-directive " +
binding.value
      })
      3. 添加实例方法
      Vue.prototype.$myMethod = function () {
         alert('vue 实例对象方法执行')
      }
   }
   window.MyPlugin = MyPlugin
})()
```

使用 vue 插件

```
Vue.use(MyPlugin)  //调用了插件内的 install 方法
```

vue-cli 目录结构介绍

```
|-- build : webpack 相关的配置文件夹(基本不需要修改)
|-- dev-server.js : 通过 express 启动后台服务器
|-- config: webpack 相关的配置文件夹(基本不需要修改)
|-- index.js: 指定的后台服务的端口号和静态资源文件夹
|-- node_modules
|-- src : 源码文件夹
|-- components: vue 组件及其相关资源文件夹
```

|-- App.vue: 应用根主组件

|-- main.js: 应用入口 js

|-- static: 静态资源文件夹

|-- .babelrc: babel 的配置文件

|-- .eslintignore: eslint 检查忽略的配置

|-- .eslintrc.js: eslint 检查的配置

|-- .gitignore: git 版本管制忽略的配置

|-- index.html: 主页面文件

|-- package.json: 应用包配置文件

|-- README.md: 应用描述说明的 readme 文件

vue 组件间通信方式

1.props

2.vue 的自定义事件

3.消息订阅与发布（如：pubsub 库）

4.slot

5.vuex

props:

父子组件间通信的基本方式

  属性值的 2 大类型:

      一般：父组件-->子组件

函数: 子组件-->父组件

隔层组件间传递: 必须逐层传递(麻烦)

兄弟组件间: 必须借助父组件(麻烦)

vue 自定义事件

子组件与父组件的通信方式

用来取代 function props

不适合隔层组件和兄弟组件间的通信

pubsub 第三方库(消息订阅与发布)

适合于任何关系的组件间通信

slot

通信是带数据的标签

注意: 标签是在父组件中解析

vuex

多组件共享状态(数据的管理)

组件间的关系也没有限制

功能比 pubsub 强大, 更适用于 vue 项目

vue-router 相关 API 说明

1. VueRouter(): 用于创建路由器的构造函数

```
new VueRouter({
    //多个配置项
})
```

2.路由配置

```
routes:[
    {
        //一般路由
        path:'/about',
        component:About
    },
    {
        //自动跳转路由
        path:'/',
        redirect:'/about'
    },
    {
        //嵌套路由
        path:'/home',
        component:home,
        children:[
        {
            path:'news',
            component:News
        },
        {
```

```
            path:'message',
            component:Message
        }
        ]
    },
    {
        // 携带参数的路由(读取的时候this.$route.params.id)
        {
            path:'mdetail/:id',
            component:MessageDetail
        }
    }
]
```

3.注册路由器

```
import router from './router'
new Vue({
    router(router:router 的缩写形式)
})
```

4.使用路由组件标签

①.<router-link>:用于生成路由链接

```
<router-link to='/xxx'>Go to XXX</router-link>
```

②.<router-view>:用来显示当前路由组件界面

<router-view :msg="msg"></router-view> //可传参数

③.<keep-alive>:用于缓存路由组件对象

<keep-alive>

<router-view></router-view>

</keep-alive>

5.优化路由器配置

linkActiveClass:'active' //制定选中的路由链接的 class

6.路由相关 API（路由编程式导航）

①.this.$router.push(path):相当于点击路由链接（可以返回到当前路由界面）

②.this.$router.replace(path):用新路由替换当前路由（不可以返回到当前路由界面）

③.this.$router.back():请求（返回）上一个记录路由

④.this.$router.go(-1):请求（返回）上一条记录路由

⑤.this.$router.go(1):请求下一个记录路由

vuex 的理解（状态自管理）

1.核心概念和 API

①.state

1)vuex 管理的状态对象

2)它应该是惟一的

```
const state = {
    xxx:initValue
}
```

②.mutations

1)包含多个直接更新 state 的方法（回掉函数）的对象

2)谁来触发：action 中的 commit('mutation 名称')

3)只能包含同步的代码，不能写异步代码

```
const mutations = {
    yyy(state,{data1}){
        //更新 state 的某个属性
    }
}
```

③.actions

1)包含多个事件回调函数的对象

2)通过执行：commit()来触发 commit 的调用，间接更新 state

3)谁来触发：组件中$store.dispatch('action 名称',data1)

```
//'zzz'
```

4)可以包含异步代码（定时器,ajax）

```
const actions = {
    zzz({commit,state},data1){
        commit('yyy',{data1})
    }
```

```
    }
```

④.getters

1)包含多个计算属性（get）的对象

2)谁来读取：组件中$store.getters.xxx

```
    const getters = {
        mmm(state){
            return...
        }
    }
```

⑤.modules

1)包含多个module

2)一个module是一个store的配置对象

3)与一个组件（包含有共享数据）对应

⑥.向外暴露store对象

```
export default new Vuex.Store({
    state,
    mutations,
    actions,
    getters
})
```

⑦.组件中

```
import {mapState,mapGetters,mapActions} from 'vuex'
```

```
export default {
   computed:{
      ...mapState(['xxx']),
      ...mapGetters(['mmm'])
   }
   method:mapActions(['zzz'])
}
{{xxx}} {{mmm}} @click = 'zzz(data)'
```

⑧.映射 store

```
import store from './store'
new Vue({
   store
})
```

⑨.store 对象

1)所有用 vuex 管理的组件中都多了一个属性$store, 它就是一个 store 对象

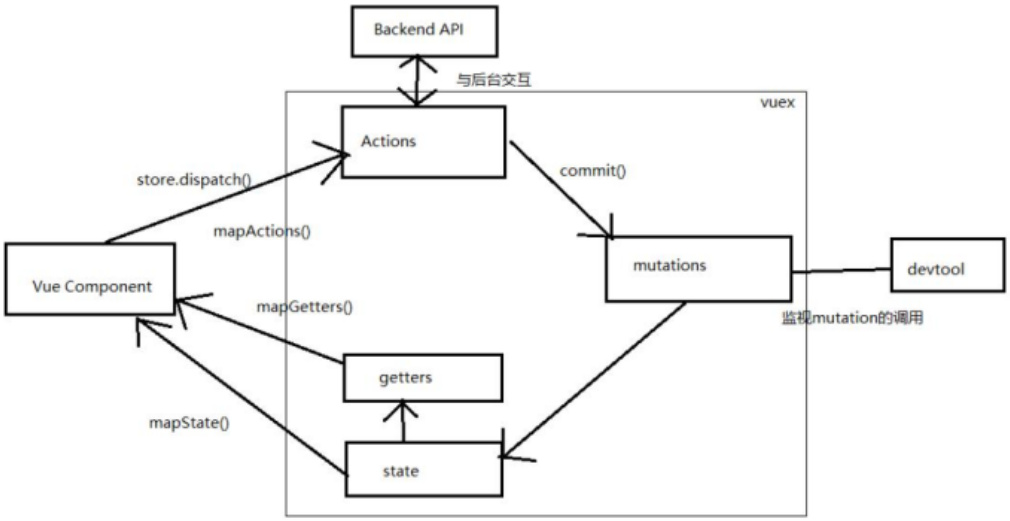
2)属性:

state:注册的 state 对象

getters:注册的 getters 对象

3)方法:

dispatch(actionName,data):分发调用 action



Vue 源码分析

准备:

1)[].slice.call(lis):将伪数组转换为真数组

2)node.nodeType:得到节点类型

3)Object.defineProperty(obj,propName,{}):给对象添加/修改属性(指定描述符)

configurable:true/false 是否可以重新 define

enumerable:true/false 是否可以枚举(for…in/keys())

value:指定初始值

writable:true/false value 是否可以修改

get:回调函数，用来得到当前属性值

set:回调函数，用来监视当前属性值的变化

4)Object.keys(obj):得到对象自身可枚举的属性名的数组

5)DocumentFragment:文档碎片(高效批量更新多个节点)

6)obj.hasOwnProperty(prop):判断 prop 是否为 obj 自身的属性

数据代理:

1) 原理: 通过一个对象代理对另一个对象（在前一个对象内部）中的属性的操作（读/写）

2) Vue 数据代理: 通过 vm 对象来代理 data 对象中所有属性的操作

3) 好处: 更方便的操作 data 中的数据

4) 基本实现流程:

a. 通过 Object.defineProperty()给 vm 添加与 data 对象的属性对应的属性描述符

b. 所有添加的属性都包含 getter/setter

c. getter/setter 内部去操作 data 中对应的属性数据

模板解析:

1） 将 el 的所有结点取出，添加到一个新建的文档 fragment 对象中

2） 对 fragment 中的所有层次子节点递归进行编译解析处理

- 对大括号表达式文本节点进行解析
- 对元素节点的指令属性进行解析
- 事件指令解析

- 一般指令解析

3) 将解析后的 fragment 添加到 el 中显示

模板解析——大括号表达式解析

1) 根据正则对象得到匹配出的表达式字符串：子匹配 /RegExp.$1 name

2) 从 data 中取出表达式对应的属性值

3) 将属性值设置为文本节点的 textContent

模板解析——事件指令解析

1) 从指令名中取出事件名

2) 根据指令的值（表达式）从 methods 中得到对应的事件处理函数对象

3) 给当前元素节点绑定指定事件名和回调函数的 dom 事件监听

4) 指令解析完后，移除此指令属性

模板解析——一般指令解析

1) 得到指令名和指令值（表达式） text/html/class

2) 从 data 中根据表达式得到对应的值

3) 根据指令名确定需要操作元素节点的什么属性

- v-text---textContent 属性
- v-html---innerHtml 属性
- v-class—className 属性

4) 将得到的表达式的值设置到对应的属性上

5） 移除元素的指令属性

数据绑定：

一旦更新了 data 中的某个属性数据，所有界面上直接使用或间接使用了此属性的节点都会更新

数据劫持：

1)数据劫持是 vue 中用来实现数据绑定的一种技术

2)基本思想是通过 defineProperty()来监视 data 中所有属性（任意层次）数据的变化，一旦变化就去更新界面

四个重要对象：

1） Observer

a. 用来给 data 所有属性数据进行劫持的构造函数

b. 给 data 中所有属性重新定义属性描述(get/set)

c. 为 data 中的所有属性创建对应的 dep 对象

2） Dep(Depend)

a. data 中的每个属性（所有层次）都对应一个 dep 对象

b. 创建的时机：

在初始化 define data 中各个属性时创建对应的 dep 对象

在 data 中的某个属性值被设置为新的对象时

c. 对象的结构

{

Id,//每个 dep 都有一个唯一的 id

Subs//包含 n 个对应 watcher 的数组（subscribes 的简写）

}

d. subs 属性说明

当 watcher 被创建时,内部将当前 watcher 对象添加到对应的 dep 对象的 subs 中

当此 data 属性的值发生改变时，subs 中所有的 watcher 都会收到更新的通知，从而最终更新对应的界面

3） Compiler

a. 用来解析模板页面的对象的构造函数（一个实例）

b. 利用 compile 对象解析模板页面

c. 每解析一个表达式(非事件指令)都会创建一个对应的 watcher 对象，并建立 watcher 与 dep 的关系

d. compile 与 watcher 关系，一对多的关系

4） watcher

a. 模板中每个非事件指令或表达式都对应一个 watcher 对象

b. 监视当前表达式数据的变化

c. 创建的时机：在初始化编译模板时

d. 对象的组成

{

vm, //vm 对象

exp, //对应指令的表达式

cb, //当表达式所对应的数据发生改变的回调函数

value, //表达式当前的值

deplds //表达式中各级属性所对应的 dep 对象的集合对象

//属性名为 dep 的 id, 属性值为 dep

}

5） 总结：dep 与 watcher 的关系：多对多

a. data 中的一个属性对应一个 dep,一个 dep 可能包含多个 watcher（模板中有几个表达式用到了同一个属性）

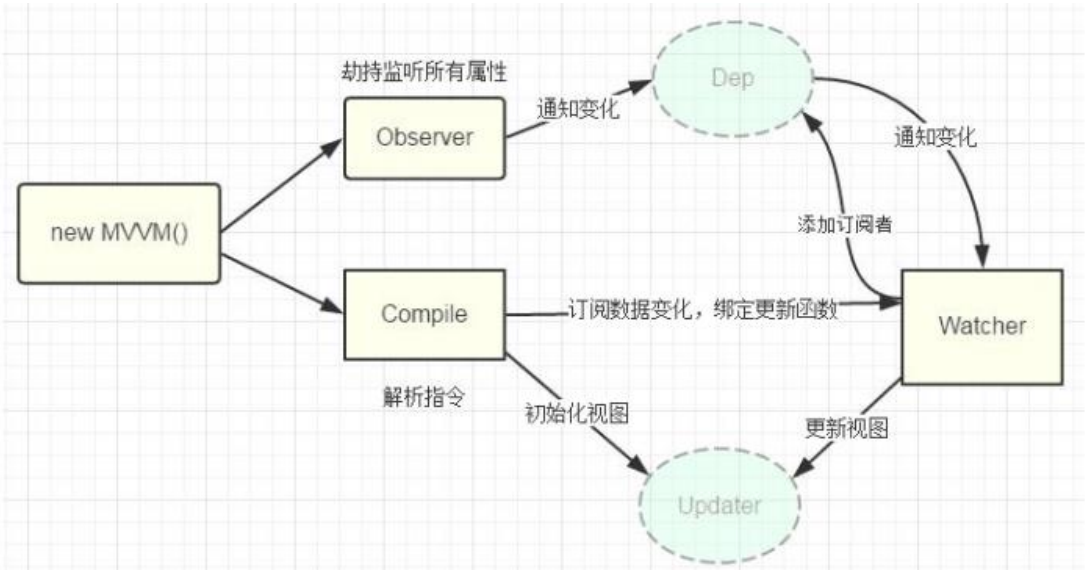
b. 模板中一个非事件表达式对应一个 watcher, 一个 watcher 中可能包含多个 dep（表达式是多层）

c. 数据绑定用到了 2 个核心技术

DefineProperty()

消息订阅与发布

MVVM 原理图分析：



双向数据绑定：

1） 双向数据绑定是建立在单向数据绑定（model➔view）的基础之上的

2） 双向数据绑定的实现原理

a. 在解析 v-model 指令时,给当前元素添加 input 监听

b. 当 input 的 value 发生改变时,将最新的值赋给当前表达式所对应的 data 属性