Vue 全家桶

第 1 章：Vue 核心

1.1. Vue 的基本认识

1.1.1. 官网

1) 英文官网: https://vuejs.org/
2) 中文官网: https://cn.vuejs.org/

1.1.2. 介绍描述

1) 渐进式 JavaScript 框架
2) 作者: 尤雨溪(一位华裔前 Google 工程师)
3) 作用: 动态构建用户界面

1.1.3. Vue 的特点

1) 遵循 MVVM 模式
2) 编码简洁, 体积小, 运行效率高, 适合移动/PC 端开发
3) 它本身只关注 UI, 可以轻松引入 vue 插件或其它第三库开发项目

1.1.4. 与其它前端 JS 框架的关联

1) 借鉴 angular 的**模板**和**数据绑定**技术
2) 借鉴 react 的**组件化**和**虚拟 DOM** 技术

1.1.5. Vue 扩展插件

1) vue-cli: vue 脚手架
2) vue-resource(axios): ajax 请求
3) vue-router: 路由
4) vuex: 状态管理
5) vue-lazyload: 图片懒加载
6) vue-scroller: 页面滑动相关
7) mint-ui: 基于 vue 的 UI 组件库(移动端)
8) element-ui: 基于 vue 的 UI 组件库(PC 端)

1.2. Vue 的基本使用

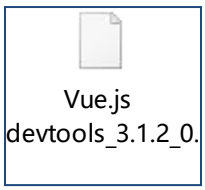
1.2.1. 效果 *(01_HelloWorld/test.html)*

1.2.2. 编码

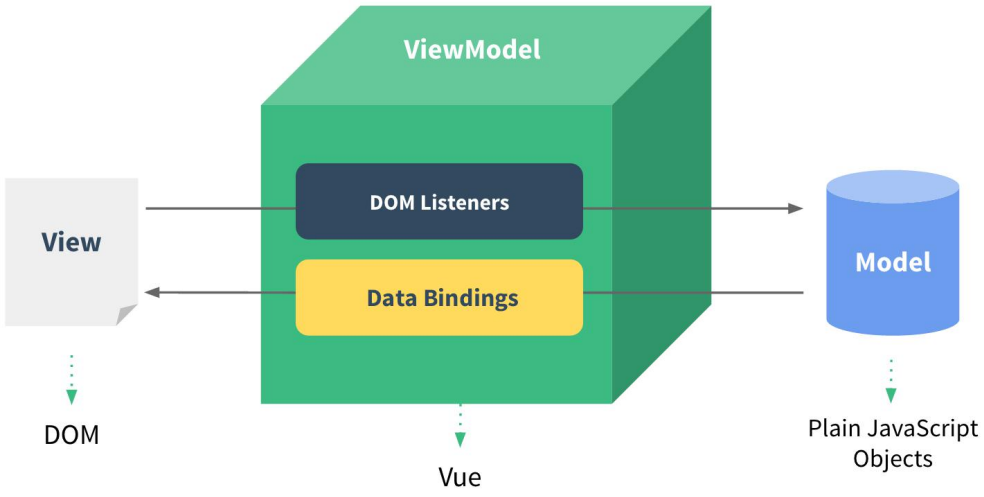
```
<div id="app">
  <input type="text" v-model="username">
  <p>Hello, {{username}}</p>
</div>

<script type="text/javascript" src="../js/vue.js"></script>
<script type="text/javascript">
  new Vue({
    el: '#app',
    data: {
      username: 'atguigu'
    }
```

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

1.2.3. 使用 vue 开发者工具调试

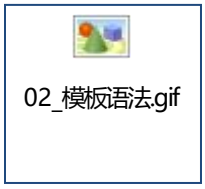


1.2.4. 理解 Vue 的 MVVM



1.3. 模板语法

1.3.1. 效果 *(02_模板语法/test.html)*



1.3.2. 模板的理解

1) 动态的 html 页面
2) 包含了一些 JS 语法代码
   a. 双大括号表达式
   b. 指令(以 v-开头的自定义标签属性)

1.3.3. 双大括号表达式

1) 语法: {{exp}}
2) 功能: 向页面输出数据
3) 可以调用对象的方法

1.3.4. 指令一: 强制数据绑定

1) 功能: 指定变化的属性值
2) 完整写法: v-bind:xxx='yyy' //yyy 会作为表达式解析执行
3) 简洁写法: :xxx='yyy'

1.3.5. 指令二: 绑定事件监听

1) 功能: 绑定指定事件名的回调函数
2) 完整写法:

   v-on:keyup='xxx'

   v-on:keyup='xxx(参数)'

   v-on:keyup.enter='xxx'
3) 简洁写法:

   @keyup='xxx'

   @keyup.enter='xxx'

1.3.6. 编码

```
<div id="app">
  <h2>1. 双大括号表达式</h2>
  <p>{{msg}}</p>
  <p>{{msg.toUpperCase()}}</p>

  <h2>2. 指令一: 强制数据绑定</h2>
  <a href="url">访问指定站点</a><br><!--不能使用-->
  <a v-bind:href="url">访问指定站点 2</a><br>
  <a :href="url">访问指定站点 3</a><br>

  <h2>3. 指令二: 绑定事件监听</h2>
  <button v-on:click="handleClick">点我</button>
  <button @click="handleClick">点我 2</button>

</div>

<script type="text/javascript" src="../js/vue.js"></script>
<script type="text/javascript">
  new Vue({
    el: '#app',
    data: {// data 的所有属性都会成功 vm 对象的属性, 而模板页面中可以直接访问
      msg: 'NBA I Love This Game!',
      url: 'http://www.baidu.com'
    },
    methods: {
      handleClick () {
        alert('处理点击')
      }
    }
  })
</script>
```

1.4. 计算属性和监视

1.4.1. 效果 *(03_计算属性和监视/test.html)*

1.4.2. 计算属性

1) 在 computed 属性对象中定义计算属性的方法

2) 在页面中使用{{方法名}}来显示计算的结果

1.4.3. 监视属性

1) 通过通过 vm 对象的$watch()或 watch 配置来监视指定的属性

2) 当属性变化时，回调函数自动调用，在函数内部进行计算

1.4.4. 计算属性高级

1) 通过 getter/setter 实现对属性数据的显示和监视

2) 计算属性存在缓存，多次读取只执行一次 getter 计算

1.4.5. 编码

```
<div id="demo">
  姓: <input type="text" placeholder="First Name" v-model="firstName"><br>
  名: <input type="text" placeholder="Last Name" v-model="lastName"><br>
  姓名 1(单向): <input type="text" placeholder="Full Name" v-model="fullName1"><br>
  姓名 2(单向): <input type="text" placeholder="Full Name" v-model="fullName2"><br>
  姓名 3(双向): <input type="text" placeholder="Full Name2" v-model="fullName3"><br>
</div>
```

```
<script type="text/javascript" src="../js/vue.js"></script>
<script type="text/javascript">
  var vm = new Vue({
    el: '#demo',
    data: {
      firstName: 'Kobe',
      lastName: 'bryant',
      fullName2: 'Kobe bryant'
    },

    computed: {

      fullName: function () {
        return this.firstName + " " + this.lastName
      },

      fullName3: {
        get: function () {
          return this.firstName + " " + this.lastName
        },
        set: function (value) {
          var names = value.split(' ')
          this.firstName = names[0]
          this.lastName = names[1]
        }
      }
    },

    watch: {
      lastName: function (newVal, oldVal) {
        this.fullName2 = this.firstName + ' ' + newVal
      }
    }
  })

  vm.$watch('firstName', function (val) {
    this.fullName2 = val + ' ' + this.lastName
  })
```

1.5. class 与 style 绑定

1.5.1. 效果 (04_class 与 style 绑定/test.html)

1.5.2. 理解

1) 在应用界面中, 某个(些)元素的样式是变化的

2) class/style 绑定就是专门用来实现动态样式效果的技术

1.5.3. class 绑定

1) :class='xxx'

2) 表达式是字符串: 'classA'

3) 表达式是对象: {classA:isA, classB: isB}

4) 表达式是数组: ['classA', 'classB']

1.5.4. style 绑定

1) :style="{ color: activeColor, fontSize: fontSize + 'px' }"

2) 其中 activeColor/fontSize 是 data 属性

1.5.5. 编码

```
<style>
  .classA {
    color: red;
  }
  .classB {
```

```
    background: blue;
  }
  .classC {
    font-size: 20px;
  }
</style>

<div id="demo">
  <h2>1. class 绑定: :class='xxx'</h2>
  <p class="classB" :class="a">表达式是字符串: 'classA'</p>
  <p :class="{classA: isA, classB: isB}">表达式是对象: {classA:isA, classB: isB}</p>
  <p :class="['classA', 'classC']"> 表达式是数组: ['classA', 'classB']</p>

  <h2>2. style 绑定</h2>
  <p :style="{color, fontSize}">style="{ color: activeColor, fontSize: fontSize +
'px' }"</p>


  <button @click="update">更新</button>
</div>

<script type="text/javascript" src="../js/vue.js"></script>
<script type="text/javascript">
  new Vue({
    el : '#demo',
    data : {
      a: 'classA',
      isA: true,
      isB: false,
      color: 'red',
      fontSize: '20px'
    },
    methods : {
      update () {
        this.a = 'classC'
        this.isA = false
        this.isB = true
        this.color =  'blue'
        this.fontSize = '30px'
      }
    }
  })
```

```
</script>
```

1.6. 条件渲染

1.6.1. 效果 (05_条件渲染/test.html)

1.6.2. 条件渲染指令

1) v-if 与 v-else

2) v-show

1.6.3. 比较 v-if 与 v-show

3) 如果需要频繁切换 v-show 较好

4) 当条件不成立时, v-if 的所有子节点不会解析(项目中使用)

1.6.4. 编码

```
<div id="demo">
  <h2 v-if="ok">表白成功</h2>
  <h2 v-else>表白失败</h2>
  <h2 v-show="ok">求婚成功</h2>
  <h2 v-show="!ok">求婚失败</h2>

  <br>
  <button @click="ok=!ok">切换</button>
</div>
```

```
<script type="text/javascript" src="../js/vue.js"></script>
<script type="text/javascript">
  var vm = new Vue({
    el: '#demo',
    data: {
      ok: false
    }
  })
</script>
```

1.7. 列表渲染

1.7.1. 效果 (06_列表渲染/test.html)

06_列表渲染.gif

06_列表的过滤和排序.gif

1) 列表显示指令

   数组: v-for / index

   对象: v-for / key

2) 列表的更新显示

   删除 item

   替换 item

3) 列表的高级处理

   列表过滤

   列表排序

1.7.2. 编码 1

```
<div id="demo">
  <h2>测试: v-for 遍历数组</h2>
  <ul>
```

```
    <li v-for="(p, index) in persons" :key="index">
      {{index}}--{{p.name}}--{{p.age}}
      --
      <button @click="deleteItem(index)">删除</button>
      --
      <button @click="updateItem(index, {name:'Jok',age:15})">更新</button>
    </li>
  </ul>

  <h2>测试: v-for 遍历对象</h2>
  <ul>
    <li v-for="(value, key) in persons[0]">
      {{ key }} : {{ value }}
    </li>
  </ul>
</div>

<script type="text/javascript" src="../js/vue.js"></script>
<script type="text/javascript">
  new Vue({
    el: '#demo',
    data: {
      persons: [
        {id: 1, name: 'Tom', age: 13},
        {id: 2, name: 'Jack', age: 12},
        {id: 3, name: 'Bob', age: 14}
      ]
    },

    methods: {
      deleteItem(index) {
        this.persons.splice(index, 1)
      },
      updateItem(index, p) {
        // this.persons[index] = p // 页面不会更新
        this.persons.splice(index, 1, p)
      }
    }
  })
</script>
```

1.7.3. 编码 2

```
<div id="demo">
  <input type="text" name="searchName" placeholder="搜索指定用户名"
v-model="searchName">
  <ul>
    <li v-for="(p, index) in filterPerson" :key="index">
      {{index}}--{{p.name}}--{{p.age}}
    </li>
  </ul>
  <button @click="setOrderType(1)">年龄升序</button>
  <button @click="setOrderType(2)">年龄降序</button>
  <button @click="setOrderType(0)">原本顺序</button>
</div>

<script type="text/javascript" src="../js/vue.js"></script>
<script type="text/javascript">
  new Vue({
    el: '#demo',
    data: {
      orderType: 0, //0 代表不排序, 1 为升序, 2 为降序
      searchName: '',
      persons: [
        {id: 1, name: 'Tom', age: 13},
        {id: 2, name: 'Jack', age: 12},
        {id: 3, name: 'Bob', age: 17},
        {id: 4, name: 'Cat', age: 14},
        {id: 4, name: 'Mike', age: 14},
        {id: 4, name: 'Monica', age: 16}
      ]
    },
    methods: {
      setOrderType (orderType) {
        this.orderType = orderType
      }
    },
    computed: {
      filterPerson() {
        let {orderType, searchName, persons} = this
        // 过滤
        persons = persons.filter(p => p.name.indexOf(searchName)!=-1)
```

```
        // 排序
        if(orderType!==0) {
          persons = persons.sort(function (p1, p2) {
            if(orderType===1) {
              return p1.age-p2.age
            } else {
              return p2.age-p1.age
            }
          })
        }
        return persons
      }
    }
  })
</script>
```

1.8. 事件处理

1.8.1. 效果 (07_事件处理/test.html)


07_事件处理.gif

1.8.2. 绑定监听:

1) v-on:xxx="fun"
2) @xxx="fun"
3) @xxx="fun(参数)"
4) 默认事件形参: event
5) 隐含属性对象: $event

1.8.3. 事件修饰符

1) .prevent：阻止事件的默认行为 event.preventDefault()

2) .stop：停止事件冒泡 event.stopPropagation()

1.8.4. 按键修饰符

1) .keycode：操作的是某个 keycode 值的键

2) .keyName：操作的某个按键名的键(少部分)

1.8.5. 编码

```
<div id="example">

  <h2>1. 绑定监听</h2>
  <button v-on:click="test1">Greet</button>
  <button @click="test1">Greet2</button>
  <button @click="test2($event, 'hello')">Greet3</button>

  <h2>2. 事件修饰符</h2>
  <!-- 阻止事件默认行为 -->
  <a href="http://www.baidu.com" @click.prevent="test3">百度一下</a>
  <br/>
  <br/>
  <!-- 停止事件冒泡 -->
  <div style="width: 200px;height: 200px;background: red" @click="test4">
    <div style="width: 100px;height: 100px;background: green"
@click.stop="test5"></div>
  </div>

  <h2>3. 按键修饰符</h2>
  <input @keyup.8="test6">
  <input @keyup.enter="test6">
</div>

<script type="text/javascript" src="../js/vue.js"></script>
<script type="text/javascript">
  new Vue({
```

```
    el: '#example',
    data: {
      name: 'Vue.js'
    },
    methods: {
      test1 (event) {
        // 方法内 `this` 指向 vm
        // alert('Hello ' + this.name + '!')
        // `event` 是原生 DOM 事件
        alert(event.target.innerHTML)
      },
      test2 (event, msg) {
        alert(event.target.innerHTML + '---' + msg)
      },

      test3() {
        alert('阻止事件的默认行为')
      },

      test4() {
        alert('out')
      },
      test5() {
        alert('inner')
      },

      test6(event) {
        alert(event.keyCode + '---' + event.target.value)
      }
    }
  })
</script>
```

1.9. 表单输入绑定

1.9.1. 效果 (08_表单输入绑定/test.html)

1.9.2. 使用 v-model 对表单数据自动收集

1) text/textarea
2) checkbox
3) radio
4) select

1.9.3. 编码

```
<div id="demo">

  <form @submit.prevent="handleSubmit">
    <span>用户名: </span>
    <input type="text" v-model="user.username"><br>

    <span>密码: </span>
    <input type="password" v-model="user.pwd"><br>

    <span>性别: </span>
    <input type="radio" id="female" value="female" v-model="user.sex">
    <label for="female">女</label>
    <input type="radio" id="male" value="male" v-model="user.sex">
    <label for="male">男</label><br>

    <span>爱好: </span>
    <input type="checkbox" id="basket" value="basketball"
v-model="user.likes">
```

```
    <label for="basket">篮球</label>
    <input type="checkbox" id="foot" value="football"
v-model="user.likes">
    <label for="foot">足球</label>
    <input type="checkbox" id="pingpang" value="pingpang"
v-model="user.likes">
    <label for="pingpang">乒乓</label><br>

    <span>城市: </span>
    <select v-model="user.cityId">
      <option value="">未选择</option>
      <option v-for="city in allCitys" :value="city.id">
        {{ city.name }}
      </option>
    </select><br>
    <span>介绍: </span>
    <textarea v-model="user.desc" rows="10"></textarea><br><br>

    <input type="submit" value="注册">
  </form>

</div>

<script type="text/javascript" src="../js/vue.js"></script>
<script type="text/javascript">
  var vm = new Vue({
    el: '#demo',
    data: {
      user: {
        username: '',
        pwd: '',
        sex: 'female',
        likes: [],
        cityId: '',
        desc: '',
      },
      allCitys: [{id: 1, name: 'BJ'}, {id: 2, name: 'SZ'},{id: 4, name:
'SH'}],
    },
    methods: {
      handleSubmit (event) {
        alert(JSON.stringify(this.user))
```

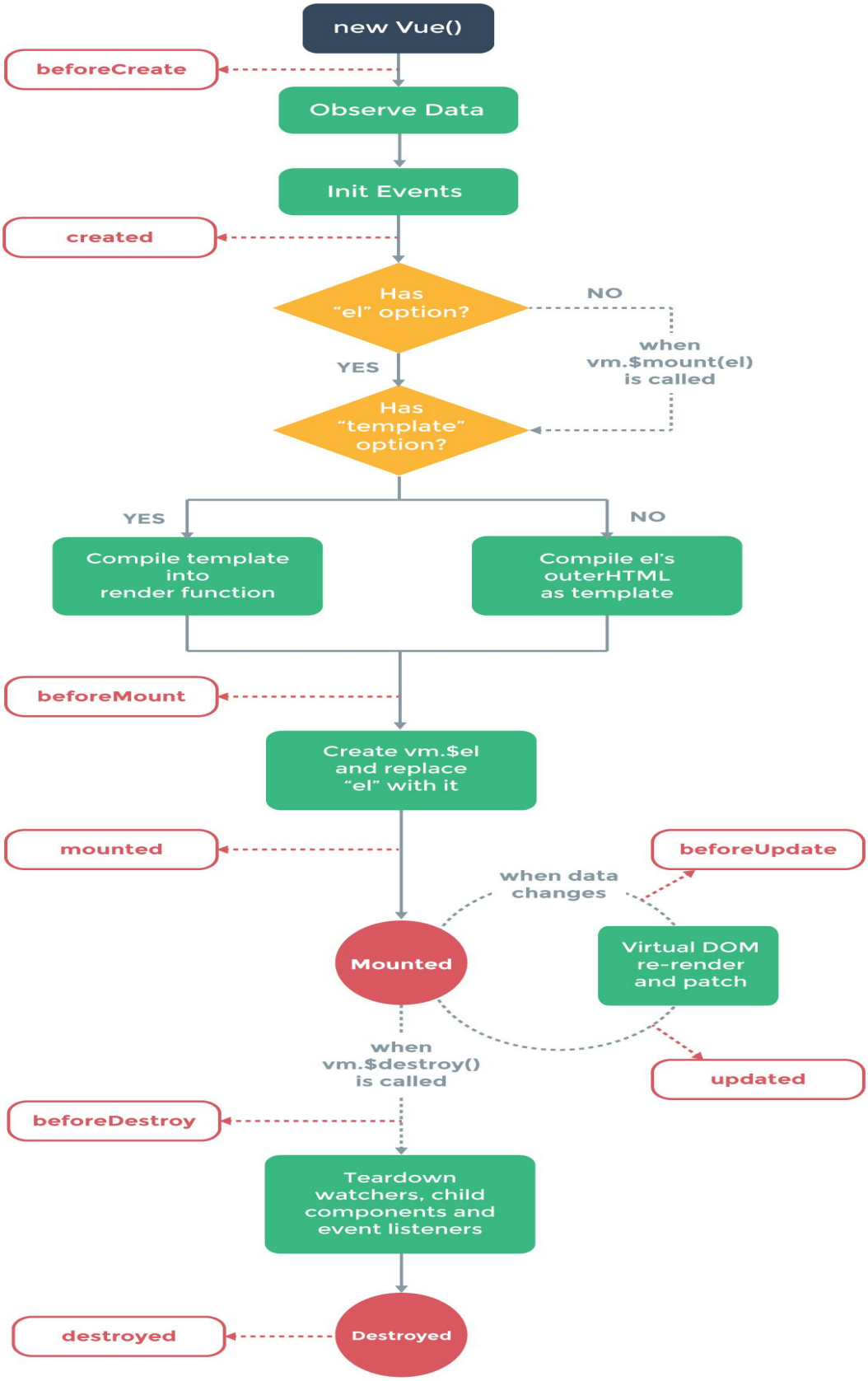
```
      }
    }
  })
</script>
```

1.10. Vue 实例生命周期

1.10.1. 效果 (09_Vue 实例_生命周期/test.html)


09_Vue实例_生命周期.gif

1.10.2. 生命周期流程图


new Vue()
beforeCreate
Observe Data
Init Events
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 vm.$destroy() is called
updated
beforeDestroy
Teardown watchers, child components and event listeners
destroyed
Destroyed

1.10.3. vue 生命周期分析

1) 初始化显示

   * *beforeCreate()*
   * *created()*
   * *beforeMount()*
   * *mounted()*

2) 更新状态: this.xxx = value

   * *beforeUpdate()*
   * *updated()*

3) 销毁 vue 实例: vm.$destory()

   * *beforeDestory()*
   * *destoryed()*

1.10.4. 常用的生命周期方法

1) created()/mounted(): 发送 ajax 请求, 启动定时器等异步任务

2) beforeDestory(): 做收尾工作, 如: 清除定时器

1.10.5. 编码

```
<div>
  <button @click="destoryVue">destory vue</button>
  <p v-show="isShowing">{{msg}}</p>
</div>

<script type="text/javascript" src="../js/vue.js"></script>
<script type="text/javascript">
  var vue = new Vue({
    el: 'div',
    data: {
      msg: '尚硅谷 IT 教育',
      isShowing: true,
      persons: []
    },
    beforeCreate () {
      console.log('beforeCreate() msg=' + this.msg)
```

```
    },
    created () {
      console.log('created() msg='+this.msg)

      this.intervalId = setInterval(() => {
        console.log('-----')
        this.isShowing = !this.isShowing
      }, 1000)
    },

    beforeMount () {
      console.log('beforeMount() msg='+this.msg)
    },
    mounted () {
      console.log('mounted() msg='+this.msg)
    },

    beforeUpdate() {
      console.log('beforeUpdate isShowing='+this.isShowing)
    },
    updated () {
      console.log('updated isShowing='+this.isShowing)
    },

    beforeDestroy () {
      console.log('beforeDestroy() msg='+this.msg)
      clearInterval(this.intervalId)
    },

    destroyed () {
      console.log('destroyed() msg='+this.msg)
    },

    methods: {
      destoryVue () {
        vue.$destroy()
      }
    }
  })
</script>
```

1.11. 过渡&动画

1.11.1. 效果 (10_过渡&动画/test.html)

10_过渡&动画1.gif

10_过渡&动画2.gif

1.11.2. vue 动画的理解

1) 操作 css 的 trasition 或 animation

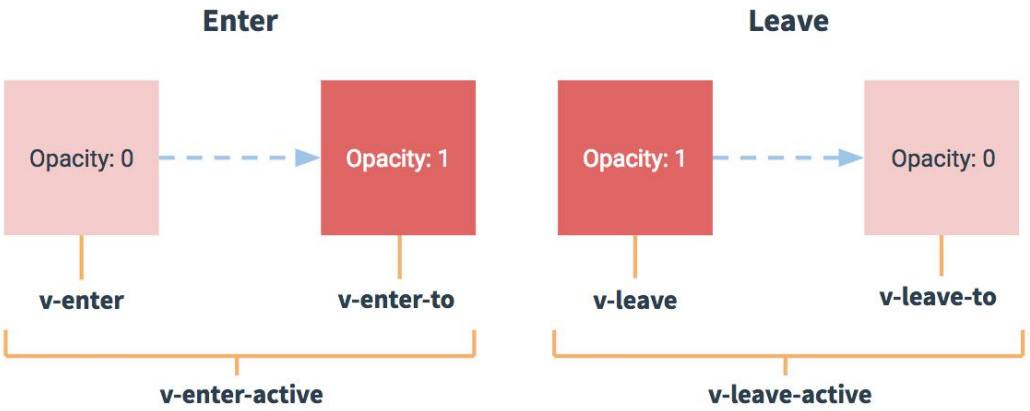
2) vue 会给目标元素添加/移除特定的 class

3) 过渡的相关类名

xxx-enter-active: 指定显示的 transition

xxx-leave-active: 指定隐藏的 transition

xxx-enter/xxx-leave-to: 指定隐藏时的样式



1.11.3. 基本过渡动画的编码

1) 在目标元素外包裹<transition name="xxx">

2) 定义 class 样式

指定过渡样式: transition

指定隐藏时的样式: opacity/其它

1.11.4. 编码 1

```
<style>
  .fade-enter-active, .fade-leave-active {
    transition: opacity .5s
  }

  .fade-enter, .fade-leave-to {
    opacity: 0
  }

  /* 可以设置不同的进入和离开动画 */
  .slide-fade-enter-active {
    transition: all .3s ease;
  }
  .slide-fade-leave-active {
    transition: all .8s cubic-bezier(1.0, 0.5, 0.8, 1.0);
  }
  .slide-fade-enter, .slide-fade-leave-to {
    transform: translateX(10px);
    opacity: 0;
  }
</style>

<div id="demo1">
  <button @click="show = !show">
    Toggle1
  </button>
  <transition name="fade">
    <p v-if="show">hello</p>
  </transition>
```

```
</div>

<div id="demo2">
  <button @click="show = !show">
    Toggle2
  </button>
  <transition name="slide-fade">
    <p v-if="show">hello</p>
  </transition>
</div>

<script type="text/javascript" src="../js/vue.js"></script>
<script type="text/javascript">
  new Vue({
    el: '#demo1',
    data: {
      show: true
    }
  })

  new Vue({
    el: '#demo2',
    data: {
      show: true
    }
  })
</script>
```

1.11.5. 编码 2

```
<style>
  .bounce-enter-active {
    animation: bounce-in .5s;
  }

  .bounce-leave-active {
    animation: bounce-in .5s reverse;
  }

  @keyframes bounce-in {
    0% {
```

```
      transform: scale(0);
    }
    50% {
      transform: scale(1.5);
    }
    100% {
      transform: scale(1);
    }
  }
</style>

<div id="test2">
  <button @click="show = !show">Toggle show</button>
  <br>
  <transition name="bounce">
    <p v-if="show" style="display: inline-block">Look at me!</p>
  </transition>
</div>

<script type="text/javascript" src="../js/vue.js"></script>
<script>
  new Vue({
    el: '#test2',
    data: {
      show: true
    }
  })
</script>
```

1.12. 过滤器

1.12.1. 效果 (11_过滤器/test.html)

1.12.2. 理解过滤器

1) 功能: 对要显示的数据进行特定格式化后再显示

2) 注意: 并没有改变原本的数据, 可是产生新的对应的数据

1.12.3. 定义和使用过滤器

1) 定义过滤器

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

2) 使用过滤器

```
<div>{{myData | filterName}}</div>
<div>{{myData | filterName(arg)}}</div>
```

1.12.4. 编码

```
<div id="test">
  <p>当前时间为: {{currentTime}}</p>
```

```
  <p>当前时间 1 为: {{currentTime | dateStr}}</p>
  <p>当前时间 2 为: {{currentTime | dateStr('YYYY-MM-DD')}}</p>
  <p>当前时间 3 为: {{currentTime | dateStr('HH:mm:ss')}}</p>
</div>

<script type="text/javascript" src="../js/vue.js"></script>
<script type="text/javascript"
src="https://cdn.bootcss.com/moment.js/2.19.0/moment.js"></script>
<script>
  // 注册过滤器
  Vue.filter('dateStr', function (value, format) {
    return moment(value).format(format || 'YYYY-MM-DD HH:mm:ss')
  })

  new Vue({
    el: '#test',
    data: {
      currentTime: new Date()
    }
  })
</script>
```

1.13. 内置指令与自定义指令

1.13.1. 效果 (12_指令/test.html)

12_指令_内置指令.gif

1.13.2. 常用内置指令

1) v:text : 更新元素的 textContent

2) v-html : 更新元素的 innerHTML

3) v-if : 如果为 true, 当前标签才会输出到页面

4) v-else: 如果为 false，当前标签才会输出到页面

5) v-show：通过控制 display 样式来控制显示/隐藏

6) v-for：遍历数组/对象

7) v-on：绑定事件监听，一般简写为@

8) v-bind：强制绑定解析表达式，可以省略 v-bind

9) v-model：双向数据绑定

10) ref：指定唯一标识, vue 对象通过$els 属性访问这个元素对象

11) v-cloak：防止闪现，与 css 配合: [v-cloak] { display: none }

1.13.3. 自定义指令

1) 注册全局指令

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

2) 注册局部指令

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

3) 使用指令

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

1.13.4. 编码 1(内置指令)

```
<style>
  [v-cloak] {
    display: none
  }
</style>

<div id="example">
```

```
  <p v-text="url"></p>
  <p v-html="url"></p>
  <img :id="myid" :src="imageSrc">
  <p>
    <span ref="message">atguigu.com</span>
    <button @click="showMsg">显示左侧文本</button>
  </p>
  <p v-cloak>{{url}}</p>
</div>

<script type="text/javascript" src="../js/vue.js"></script>
<script type="text/javascript">
  alert('模拟加载慢')
  new Vue({
    el: '#example',
    data: {
      url: '<a href="http://www.atguigu.com">尚硅谷</a>',
      myid: 'abc123',
      imageSrc: 'http://cn.vuejs.org/images/logo.png'
    },

    methods: {
      showMsg: function () {
        alert(this.$refs.message.textContent)
      }
    }
  })
</script>
```

1.13.5. 编码 2(自定义指令)

需求: 自定义 2 个指令

1. 功能类型于 v-text, 但转换为全大写
2. 功能类型于 v-text, 但转换为全小写

```
<div id="demo1">
  <p v-upper-text="msg"></p>
  <p v-lower-text="msg"></p>
</div>

<div id="demo2">
```

```
  <p v-upper-text="msg2"></p>
  <p v-lower-text="msg2"></p> <!--局部指令，此处不能使用-->
</div>

<script type="text/javascript" src="../js/vue.js"></script>
<script type="text/javascript">
  //注册全局指令
  Vue.directive('upper-text', function (el, binding) {
    el.innerHTML = binding.value.toUpperCase()
  })

  new Vue({
    el: '#demo1',
    data: {
      msg: 'NBA love this game!'
    },
    directives: { // 注册局部指令
      'lower-text': {
        bind (el, binding) {
          el.innerHTML = binding.value.toLowerCase()
        }
      }
    }
  })

  new Vue({
    el: '#demo2',
    data: {
      msg2: 'I Like You'
    }
  })
</script>
```

1.14. 自定义插件

1.14.1. 效果 (13_插件/test.html)

13_插件.gif

1.14.2. 说明

1) Vue 插件是一个包含 install 方法的对象

2) 通过 install 方法给 Vue 或 Vue 实例添加方法，定义全局指令等

1.14.3. 编码

1) 插件 JS

```
/**
 * 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
})()
```

2) 页面使用插件

```
<div id="demo">
  <!--使用自定义指令-->
  <p v-my-directive="msg"></p>
</div>

<script type="text/javascript" src="../js/vue.js"></script>
<script type="text/javascript" src="vue-myPlugin.js"></script>
<script type="text/javascript">

  //使用自定义插件
  Vue.use(MyPlugin)

  var vm = new Vue({
    el: '#demo',
    data: {
      msg: 'atguigu'
    }
  })
  //调用自定义的静态方法
  Vue.myGlobalMethod()
  //调用自定义的对象方法
  vm.$myMethod()
</script>
```

第 2 章：vue 组件化编码

2.1. 使用 vue-cli 创建模板项目

2.1.1. 说明

1) vue-cli 是 vue 官方提供的脚手架工具

2) github: https://github.com/vuejs/vue-cli

3) 作用：从 https://github.com/vuejs-templates 下载模板项目

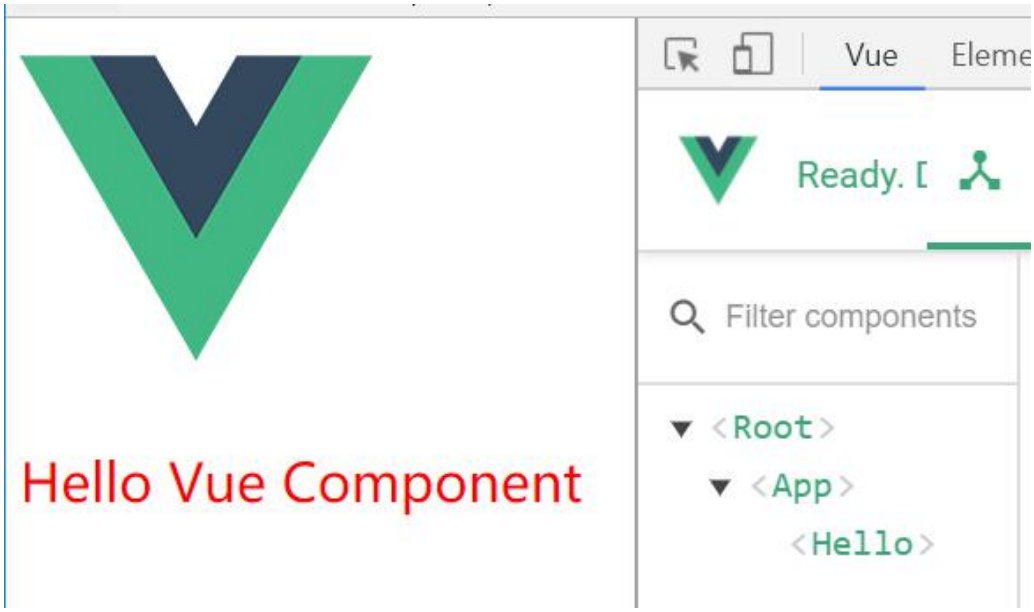
2.1.2. 创建 vue 项目

npm install -g vue-cli
vue init webpack vue_demo
cd vue_demo
npm install
npm run dev
访问: http://localhost:8080/

2.1.3. 模板项目的结构

|-- 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

2.1.4. 效果



2.2. 项目的打包与发布

2.2.1. 打包:

npm run build

2.2.2. 发布 1: 使用静态服务器工具包

npm install -g serve
serve dist
访问: http://localhost:5000

2.2.3. 发布 2: 使用动态 web 服务器(tomcat)

修改配置: webpack.prod.conf.js

```
output: {
  publicPath: '/xxx/'    //打包文件夹的名称
}
```

重新打包:

npm run build

修改 dist 文件夹为项目名称: xxx
将 xxx 拷贝到运行的 tomcat 的 webapps 目录下
访问: http://localhost:8080/xxx

2.3. eslint

2.3.1. 说明

1) ESLint 是一个代码规范检查工具
2) 它定义了很多特定的规则，一旦你的代码违背了某一规则，eslint 会作出非常有用的提示
3) 官网: http://eslint.org/
4) 基本已替代以前的 JSLint

2.3.2. ESLint 提供以下支持

1) ES
2) JSX
3) style 检查
4) 自定义错误和提示

2.3.3. ESLint 提供以下几种校验

1) 语法错误校验
2) 不重要或丢失的标点符号，如分号
3) 没法运行到的代码块（使用过 WebStorm 的童鞋应该了解）
4) 未被使用的参数提醒
5) 确保样式的统一规则，如 sass 或者 less
6) 检查变量的命名

2.3.4. 规则的错误等级有三种

1) 0：关闭规则。
2) 1：打开规则，并且作为一个警告（信息打印黄色字体）
3) 2：打开规则，并且作为一个错误（信息打印红色字体）

2.3.5. 相关配置文件

1) .eslintrc.js : 全局规则配置文件

```
'rules': {
    'no-new': 1
}
```

2) 在 js/vue 文件中修改局部规则

```
/* eslint-disable no-new */
new Vue({
    el: 'body',
    components: { App }
})
```

3) .eslintignore: 指令检查忽略的文件

```
*.js
*.vue
```

2.4. 组件定义与使用

2.4.1. vue 文件的组成(3 个部分)

1) 模板页面

```
<template>
    页面模板
</template>
```

2) JS 模块对象

```
<script>
    export default {
        data() {return {}},
        methods: {},
        computed: {},
        components: {}
    }
</script>
```

3) 样式

```
<style>
```

样式定义
</style>

2.4.2. 基本使用

1) 引入组件

2) 映射成标签

3) 使用组件标签

```
<template>
  <HelloWorld></HelloWorld>
  <hello-world></hello-world>
</template>
<script>
  import HelloWorld from './components/HelloWorld'
  export default {
    components: {
      HelloWorld
    }
  }
</script>
```

2.4.3. 关于标签名与标签属性名书写问题

1) 写法一: 一模一样

2) 写法二: 大写变小写, 并用-连接

2.5. 组件间通信

2.5.1. 组件间通信基本原则

1) 不要在子组件中直接修改父组件的状态数据
2) 数据在哪, 更新数据的行为(函数)就应该定义在哪

2.5.2. 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(后面单独讲)

2.6. 组件间通信 1: props

2.6.1. 使用组件标签时

```
<my-component name='tom' :age='3' :set-name='setName'></my-component>
```

2.6.2. 定义 MyComponent 时

1) 在组件内声明所有的 props
2) 方式一: 只指定名称

   ```
   props: ['name', 'age', 'setName']
   ```

3) 方式二: 指定名称和类型

   ```
   props: {
     name: String,
     age: Number,
     setNmae: Function
   }
   ```

4) 方式三: 指定名称/类型/必要性/默认值

```
props: {
    name: {type: String, required: true, default:xxx},
}
```

2.6.3. 注意

1) 此方式用于父组件向子组件传递数据
2) 所有标签属性都会成为组件对象的属性, 模板页面可以直接引用
3) 问题:
   a. 如果需要向非子后代传递数据必须多层逐层传递
   b. 兄弟组件间也不能直接 props 通信, 必须借助父组件才可以

2.7. 组件间通信 2: vue 自定义事件

2.7.1. 绑定事件监听

```
// 方式一: 通过 v-on 绑定
@delete_todo="deleteTodo"
// 方式二: 通过$on()
this.$refs.xxx.$on('delete_todo', function (todo) {
    this.deleteTodo(todo)
})
```

2.7.2. 触发事件

```
// 触发事件(只能在父组件中接收)
this.$emit(eventName, data)
```

2.7.3. 注意:

1) 此方式只用于子组件向父组件发送消息(数据)
2) 问题: 隔代组件或兄弟组件间通信此种方式不合适

2.8. 组件间通信 3: 消息订阅与发布(PubSubJS 库)

2.8.1. 订阅消息

PubSub.subscribe('msg', function(msg, data){})

2.8.2. 发布消息

PubSub.publish('msg', data)

2.8.3. 注意

1) 优点: 此方式可实现任意关系组件间通信(数据)

2.8.4. 事件的 2 个重要操作(总结)

1) 绑定事件监听 (订阅消息)

目标: 标签元素 <button>
事件名(类型): click/focus
回调函数: function(event){}

2) 触发事件 (发布消息)

DOM 事件: 用户在浏览器上对应的界面上做对应的操作
自定义: 编码手动触发

2.9. 组件间通信 4: slot

2.9.1. 理解

此方式用于父组件向子组件传递`标签数据`

2.9.2. 子组件: Child.vue

```
<template>
    <div>
```

```
            <slot name="xxx">不确定的标签结构 1</slot>
            <div>组件确定的标签结构</div>
            <slot name="yyy">不确定的标签结构 2</slot>
        </div>
</template>
```

2.9.3. 父组件: Parent.vue

```
<child>
        <div slot="xxx">xxx 对应的标签结构</div>
        <div slot="yyy">yyyy 对应的标签结构</div>
</child>
```

2.10. demo1: comment manage

2.11. demo2: todo list

第 3 章：vue-ajax

3.1. vue 项目中常用的 2 个 ajax 库

3.1.1. vue-resource

vue 插件，非官方库, vue1.x 使用广泛

3.1.2. axios

通用的 ajax 请求库，官方推荐, vue2.x 使用广泛

3.2. vue-resource 的使用

3.2.1. 在线文档

https://github.com/pagekit/vue-resource/blob/develop/docs/http.md

3.2.2. 下载

npm install vue-resource --save

3.2.3. 编码

```
// 引入模块
import VueResource from 'vue-resource'
// 使用插件
Vue.use(VueResource)

// 通过 vue/组件对象发送 ajax 请求
this.$http.get('/someUrl').then((response) => {
   // success callback
   console.log(response.data) //返回结果数据
}, (response) => {
   // error callback
```

```
    console.log(response.statusText) //错误信息
})
```

3.3. axios 的使用

3.3.1. 效果

3.2. 在线文档

https://github.com/pagekit/vue-resource/blob/develop/docs/http.md

3.3. 下载:

npm install axios --save

3.4. 编码

```
// 引入模块
import axios from 'axios'

// 发送 ajax 请求
axios.get(url)
    .then(response => {
        console.log(response.data) // 得到返回结果数据
    })
    .catch(error => {
        console.log(error.message)
    })
```

3.4. 测试接口

接口 1: https://api.github.com/search/repositories?q=v&sort=stars
接口 2: https://api.github.com/search/users?q=aa

3.5. demo3: github users

第 4 章：vue UI 组件库

4.1. 常用

1) Mint UI:
   a. 主页: http://mint-ui.github.io/#!/zh-cn
   b. 说明: 饿了么开源的基于 vue 的移动端 UI 组件库
2) Element
   a. 主页: http://element-cn.eleme.io/#/zh-CN
   b. 说明: 饿了么开源的基于 vue 的 PC 端 UI 组件库

4.2. 使用 Mint UI

4.2.1. 下载:

```
npm install --save mint-ui
```

4.2.2. 实现按需打包

1. 下载

npm install --save-dev babel-plugin-component

2. 修改 babel 配置

```
"plugins": ["transform-runtime",["component", [
  {
    "libraryName": "mint-ui",
    "style": true
  }
]]]
```

4.2.3. mint-ui 组件分类

1) 标签组件

2) 非标签组件

4.2.4. 使用 mint-ui 的组件

1) index.html

```
<meta name="viewport" content="width=device-width, initial-scale=1, maximum-scale=1, minimum-scale=1, user-scalable=no" />

<script src="https://as.alipayobjects.com/g/component/fastclick/1.0.6/fastclick.js"></script>
<script>
  if ('addEventListener' in document) {
    document.addEventListener('DOMContentLoaded', function() {
      FastClick.attach(document.body);
    }, false);
  }
  if(!window.Promise) {
    document.writeln('<script src="https://as.alipayobjects.com/g/component/es6-promise/3.2.2/es6-promise.min.js"'+'>'+'<'+'/'+'script>');
  }
</script>
```

2) main.js

```
import {Button} from 'mint-ui'
Vue.component(Button.name, Button)
```

3) App.vue

```
<template>
  <mt-button @click="handleClick" type="primary" style="width: 100%">Test</mt-button>
</template>

<script>
  import {Toast} from 'mint-ui'
  export default {
    methods: {
      handleClick () {
        Toast('点击了测试');
      }
    }
  }
</script>
```

第 5 章：vue-router

5.1. 理解

5.1.1. 说明

1) 官方提供的用来实现 SPA 的 vue 插件

2) github: https://github.com/vuejs/vue-router

3) 中文文档: http://router.vuejs.org/zh-cn/

4) 下载: npm install vue-router --save

5.1.2. 相关 API 说明

1) VueRouter(): 用于创建路由器的构建函数

```
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'
  }
]
```

3) 注册路由器

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

4) 使用路由组件标签

1. <router-link>: 用来生成路由链接

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

5.2. 基本路由

5.2.1. 效果

基本路由.gif

5.2.2. 路由组件

Home.vue
About.vue

5.2.3. 应用组件: App.vue

```
<div>
        <!--路由链接-->
        <router-link to="/about">About</router-link>
        <router-link to="/home">Home</router-link>
        <!--用于渲染当前路由组件-->
        <router-view></router-view>
</div>
```

5.2.4. 路由器模块: src/router/index.js

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

5.2.5. 注册路由器: main.js

```
import Vue from 'vue'
import router from './router'
// 创建 vue 配置路由器
new Vue({
    el: '#app',
    router,
    render: h => h(app)
})
```

5.2.6. 优化路由器配置

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

5.2.7. 总结: 编写使用路由的 3 步

1) 定义路由组件
2) 注册路由
3) 使用路由

<router-link>
<router-view>

5.3. 嵌套路由

5.3.1. 效果

嵌套路由.gif

5.3.2. 子路由组件

News.vue
Message.vue

5.3.3. 配置嵌套路由: router.js

```
path: '/home',
component: home,
children: [
  {
    path: 'news',
    component: News
  },
  {
    path: 'message',
    component: Message
  }
]
```

5.3.4. 路由链接: Home.vue

```
<router-link to="/home/news">News</router-link>
<router-link to="/home/message">Message</router-link>
<router-view></route-view>
```

5.4. 向路由组件传递数据

5.4.1. 效果

向路由组件传递数据.gif

5.4.2. 方式 1: 路由路径携带参数(param/query)

1) 配置路由

```
children: [
  {
    path: 'mdetail/:id',
    component: MessageDetail
  }
]
```

2) 路由路径

```
<router-link :to="'/home/message/mdetail/'+m.id">{{m.title}}</router-link>
```

3) 路由组件中读取请求参数

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

5.4.3. 方式 2: <router-view>属性携带数据

```
<router-view :msg="msg"></router-view>
```

5.5. 缓存路由组件对象

5.5.1. 理解

1) 默认情况下, 被切换的路由组件对象会死亡释放, 再次回来时是重新创建的
2) 如果可以缓存路由组件对象, 可以提高用户体验

5.5.2. 编码实现

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

5.6. 编程式路由导航

5.6.1. 效果

编程导航.gif

5.6.2. 相关 API

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)上一个记录路由
5) this.$router.go(1): 请求下一个记录路由

第 6 章：vuex

6.1. vuex 理解

6.1.1. vuex 是什么

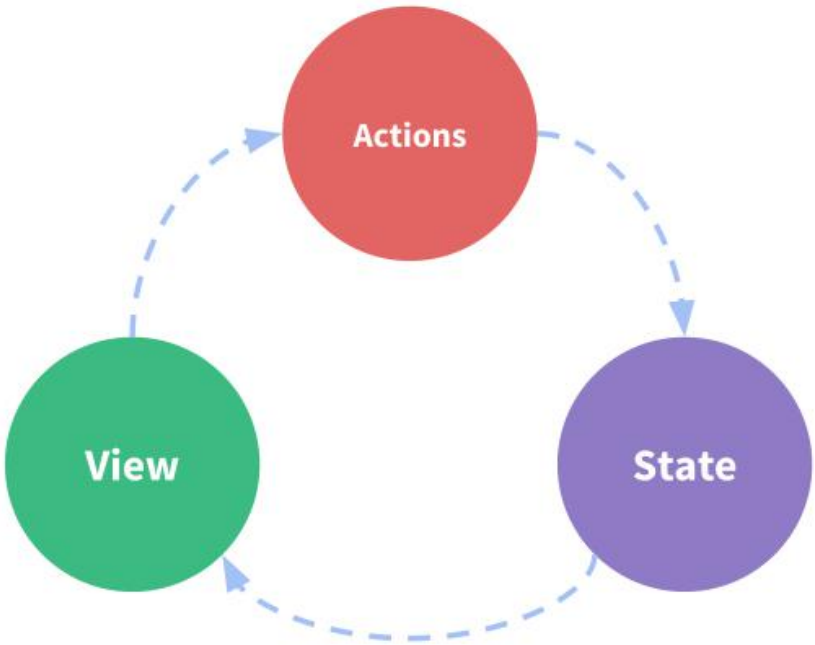
1) github 站点: https://github.com/vuejs/vuex
2) 在线文档: https://vuex.vuejs.org/zh-cn/
3) 简单来说: 对 vue 应用中多个组件的共享状态进行集中式的管理(读/写)

6.1.2. 状态自管理应用

1) state: 驱动应用的数据源

2) view: 以声明方式将 state 映射到视图

3) actions: 响应在 view 上的用户输入导致的状态变化(包含 n 个更新状态的方法)



6.1.3. 多组件共享状态的问题

1) 多个视图依赖于同一状态

2) 来自不同视图的行为需要变更同一状态

3) 以前的解决办法

   a. 将数据以及操作数据的行为都定义在父组件

   b. 将数据以及操作数据的行为传递给需要的各个子组件(有可能需要多级传递)

4) vuex 就是用来解决这个问题的

6.2. vuex 核心概念和 API

6.2.1. state

1) vuex 管理的状态对象

2) 它应该是唯一的

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

6.2.2. mutations

1) 包含多个直接更新 state 的方法(回调函数)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
    }
}
```

6.2.3. actions

1) 包含多个事件回调函数的对象

2) 通过执行: commit()来触发 mutation 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})
    }
}
```

6.2.4. 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 ...
```

```
    }
}
```

6.2.5. 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

6.2.6. 向外暴露 store 对象

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

6.2.7. 组件中

```
import {mapState, mapGetters, mapActions} from 'vuex'
export default {
    computed: {
        ...mapState(['xxx']),
        ...mapGetters(['mmm']),
    }
    methods: mapActions(['zzz'])
}

{{xxx}} {{mmm}} @click="zzz(data)"
```

6.2.8. 映射 store

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

6.2.9. 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

6.3. demo1: 计数器

counter.gif

5.3.1. store.js

```
/**
 * vuex 的 store 对象模块
 */
import Vue from 'vue'
import Vuex from 'vuex'

Vue.use(Vuex)

/*
state 对象
  类似于 data
 */
const state = {
  count: 0   // 初始化状态数据
}

/*
 mutations 对象
```

```
  包含个方法: 能直接更新 state
  一个方法就是一个 mutation
  mutation 只能包含更新 state 的同步代码, 也不会有逻辑
  mutation 由 action 触发调用: commit('mutationName')
 */
const mutations = {
  INCREMENT(state) {
    state.count++
  },
  DECREMENT (state) { // ctrl + shift + x
    state.count--
  }
}

/*
actions 对象
 包含个方法: 触发 mutation 调用, 间接更新 state
 一个方法就是一个 action
 action 中可以有逻辑代码和异步代码
 action 由组件来触发调用: this.$store.dispatch('actionName')
 */
const actions = {
  increment ({commit}) {
    commit('INCREMENT')
  },

  decrement ({commit}) {
    commit('DECREMENT')
  },

  incrementIfOdd ({commit, state}) {
    if(state.count%2===1) {
      commit('INCREMENT')
    }
  },

  incrementAsync ({commit}) {
    setTimeout(() => {
      commit('INCREMENT')
    }, 1000)
  }
}
```

```
/*
getters 对象
  包含多个get 计算计算属性方法
 */
const getters = {
  oddOrEven (state) {
    return state.count%2===0 ? '偶数' : '奇数'
  },
  count (state) {
    return state.count
  }
}

// 向外暴露 store 实例对象
export default new Vuex.Store({
  state,
  mutations,
  actions,
  getters
})
```

6.3.2. main.js

```
import Vue from 'vue'
import app from './app1.vue'
// import app from './app.vue'
import store from './store'

new Vue({
  el: '#app',
  render: h => h(app),
  store  // 所有组件都多个一个属性: $store
})
```

6.3.3. app.vue(未优化前)

```
<template>
```

```
  <div>
    <p>clicked: {{$store.state.count}} times, count is {{oddOrEven}}</p>
    <button @click="increment">+</button>
    <button @click="decrement">-</button>
    <button @click="incrementIfOdd">increment if odd</button>
    <button @click="incrementAsync">increment async</button>
  </div>
</template>

<script>
  export default {
    computed: {
      oddOrEven () {
        return this.$store.getters.oddOrEven
      }
    },

    methods: {
      increment () {
        this.$store.dispatch('increment')
      },
      decrement () {
        this.$store.dispatch('decrement')
      },
      incrementIfOdd () {
        this.$store.dispatch('incrementIfOdd')
      },
      incrementAsync () {
        this.$store.dispatch('incrementAsync')
      }
    }
  }
</script>

<style>

</style>
```

6.3.4. app2.vue(优化后)

```
<template>
  <div>
    <p>clicked: {{count}} times, count is {{oddOrEven2}}</p>
    <button @click="increment">+</button>
    <button @click="decrement">-</button>
    <button @click="incrementIfOdd">increment if odd</button>
    <button @click="incrementAsync">increment async</button>
  </div>
</template>

<script>
  import {mapGetters, mapActions} from 'vuex'

  export default {
    computed: mapGetters({  // 名称不一样
      oddOrEven2: 'oddOrEven',
      count: 'count'
    }),

    methods: mapActions(['increment', 'decrement', 'incrementIfOdd',
'incrementAsync']) // 名称一样
  }
</script>

<style>

</style>
```

6.4. demo2: todo list

todo list.gif

6.3.1. store/types.js

```
/**
 * 包含多个mutation name
 */
export const RECEIVE_TODOS = 'receive_todos'
export const ADD_TODO = 'add_todo'
export const REMOVE_TODO = 'remove_todo'
export const DELETE_DONE = 'delete_done'
export const UPDATE_ALL_TODOS = 'update_all_todos'
```

6.3.2. store/mutations.js

```
import {RECEIVE_TODOS, ADD_TODO, REMOVE_TODO, DELETE_DONE, UPDATE_ALL_TODOS} from
'./types'

export default {
  [RECEIVE_TODOS] (state, {todos}) {
    state.todos = todos
  },

  [ADD_TODO] (state, {todo}) {
    state.todos.unshift(todo)
  },

  [REMOVE_TODO] (state, {index}) {
    state.todos.splice(index, 1)
  },

  [DELETE_DONE] (state) {
    state.todos = state.todos.filter(todo => !todo.complete)
  },

  [UPDATE_ALL_TODOS] (state, {isCheck}) {
    state.todos.forEach(todo => todo.complete = isCheck)
  }
}
```

6.3.3. store/actions.js

```
import storageUtil from '../util/storageUtil'
import {RECEIVE_TODOS, ADD_TODO, REMOVE_TODO, DELETE_DONE, UPDATE_ALL_TODOS} from
'./types'

export default {
  readTodo ({commit}) {
    setTimeout(() => {
      const todos = storageUtil.fetch()
      // 提交commit 触发mutation 调用
      commit(RECEIVE_TODOS, {todos})
    }, 1000)
  },

  addTodo ({commit}, todo) {
    commit(ADD_TODO, {todo})
  },

  removeTodo ({commit}, index) {
    commit(REMOVE_TODO, {index})
  },

  deleteDone ({commit}) {
    commit(DELETE_DONE)
  },

  updateAllTodos ({commit}, isCheck) {
    commit(UPDATE_ALL_TODOS, {isCheck})
  }
}
```

6.3.4. store/getters.js

```
export default {
  todos (state) {
    return state.todos
  },
```

```
  totalSize (state) {
    return state.todos.length
  },

  completeSize (state) {
    return state.todos.reduce((preTotal, todo) => {
      return preTotal + (todo.complete ? 1 : 0)
    }, 0)
  },

  isAllComplete (state, getters) {
    return  getters.totalSize===getters.completeSize && getters.totalSize>0
  }
}
```

6.3.5. store/index.js

```
import Vue from 'vue'
import Vuex from 'vuex'
import mutations from './mutations'
import actions from './actions'
import getters from './getters'

Vue.use(Vuex)

const state = {
  todos: []
}

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

6.3.6. components/app.vue

```
<template>
  <div class="todo-container">
    <div class="todo-wrap">
      <todo-header></todo-header>
      <todo-main></todo-main>
      <todo-footer></todo-footer>
    </div>
  </div>
</template>

<script>
  import todoHeader from './todoHeader.vue'
  import todoMain from './todoMain.vue'
  import todoFooter from './todoFooter.vue'
  import storageUtil from '../util/storageUtil'

  export default {
    created () {
      // 模拟异步读取数据
      this.$store.dispatch('readTodo')
    },

    components: {
      todoHeader,
      todoMain,
      todoFooter
    }
  }
</script>

<style>
  .todo-container {
    width: 600px;
    margin: 0 auto;
  }

  .todo-container .todo-wrap {
    padding: 10px;
    border: 1px solid #ddd;
```

```
    border-radius: 5px;
  }
</style>
```

6.3.7. components/todoHeader.vue

```
<template>
  <div class="todo-header">
    <input type="text" placeholder="请输入你的任务名称，按回车键确认"
      v-model="title" @keyup.enter="addItem"/>
  </div>
</template>

<script type="text/ecmascript-6">
  export default {
    data () {
      return {
        title: null
      }
    },
    methods: {
      addItem () {
        const title = this.title && this.title.trim()
        if (title) {
          const todo = {
            title,
            complete: false
          }
          this.$store.dispatch('addTodo', todo)
          this.title = null
        }
      }
    }
  }
</script>

<style>
  .todo-header input {
    width: 560px;
    height: 28px;
```

```
    font-size: 14px;
    border: 1px solid #ccc;
    border-radius: 4px;
    padding: 4px 7px;
  }

  .todo-header input:focus {
    outline: none;
    border-color: rgba(82, 168, 236, 0.8);
    box-shadow: inset 0 1px 1px rgba(0, 0, 0, 0.075), 0 0 8px rgba(82, 168, 236, 0.6);
  }
</style>
```

6.3.8. components/todoMain.vue

```
<template>
  <ul class="todo-main">
    <todo-item v-for="(todo, index) in
todos" :todo="todo" :key="index" :index="index"></todo-item>
  </ul>
</template>

<script type="text/ecmascript-6">
  import todoItem from './todoItem'
  import storageUtil from '../util/storageUtil'

  export default {

    components: {
      todoItem
    },

    computed: {
      todos () {
        return this.$store.getters.todos
      }
    },

    watch: {
      todos: {// 深度监视todos, 一旦有变化立即保存
```

```
        handler: storageUtil.save,
        deep: true
      }
    },
  }
</script>

<style>
  .todo-main {
    margin-left: 0px;
    border: 1px solid #ddd;
    border-radius: 2px;
    padding: 0px;
  }

  .todo-empty {
    height: 40px;
    line-height: 40px;
    border: 1px solid #ddd;
    border-radius: 2px;
    padding-left: 5px;
    margin-top: 10px;
  }
</style>
```

6.3.9. components/todoItem.vue

```
<template>
  <li :style="{background: libg}"
      @mouseenter="handleStyle(true)" @mouseleave="handleStyle(false)">
    <label>
      <input type="checkbox" v-model="todo.complete"/>
      <span>{{todo.title}}</span>
    </label>
    <button class="btn btn-danger" v-show="isShown" @click="deleteItem">删除</button>
  </li>
</template>

<script type="text/ecmascript-6">
  export default {
```

```
    props: ['todo', 'index'],
    data () {
      return {
        isShown: false,
        libg: '#fff'
      }
    },
    methods: {
      handleStyle (isEnter) {
        if (isEnter) {
          this.isShown = true
          this.libg = '#ddd'
        } else {
          this.isShown = false
          this.libg = '#fff'
        }
      },
      deleteItem () {
        const {todo, deleteTodo, index} = this
        if (window.confirm(`确定删除${todo.title}的评论吗?`)) {
          // deleteTodo(index)
          this.$store.dispatch('removeTodo', index)
        }
      }
    }
  }
</script>

<style>
  li {
    list-style: none;
    height: 36px;
    line-height: 36px;
    padding: 0 5px;
    border-bottom: 1px solid #ddd;
  }

  li label {
    float: left;
    cursor: pointer;
  }
```

```
  li label li input {
    vertical-align: middle;
    margin-right: 6px;
    position: relative;
    top: -1px;
  }

  li button {
    float: right;
    display: none;
    margin-top: 3px;
  }

  li:before {
    content: initial;
  }

  li:last-child {
    border-bottom: none;
  }
</style>
```

6.3.10. components/todoFooter.vue

```
<template>
  <div class="todo-footer">
    <label>
      <input type="checkbox" v-model="isAllDone"/>
    </label>
    <span>
      <span>已完成{{completeSize}}</span> / 全部{{totalSize}}
    </span>
    <button class="btn btn-danger" @click="deleteDone" v-show="completeSize>0">清除已
完成任务</button>
  </div>
</template>

<script>
  import {mapGetters, mapActions} from 'vuex'
  export default {
```

```
    methods: mapActions(['deleteDone']),

    computed: {
      isAllDone: {
        get () {
          return this.$store.getters.isAllComplete
        },
        set (value) {
          //this.$emit('updateTodos', value)
          this.$store.dispatch('updateAllTodos', value)
        }
      },
      ...mapGetters(['completeSize', 'totalSize'])
    }
  }
/*
  const arr1 = [1, 3, 5]
  const arr2 = [4, ...arr1, 7]
  const obj = {
    a: 1,
    b () {

    }
  }
  const obj2 = {
    c: 3,
    ...obj
  }*/
</script>

<style>
  .todo-footer {
    height: 40px;
    line-height: 40px;
    padding-left: 6px;
    margin-top: 5px;
  }

  .todo-footer label {
    display: inline-block;
    margin-right: 20px;
```

```
    cursor: pointer;
  }

  .todo-footer label input {
    position: relative;
    top: -1px;
    vertical-align: middle;
    margin-right: 5px;
  }

  .todo-footer button {
    float: right;
    margin-top: 5px;
  }
</style>
```

6.3.11. util/storageUtil.js

```
var STORAGE_KEY = 'todos'
export default {
  fetch () {
    return JSON.parse(localStorage.getItem(STORAGE_KEY) || '[]')
  },
  save (todos) {
    localStorage.setItem(STORAGE_KEY, JSON.stringify(todos))
  }
}
```

6.3.12. base.css

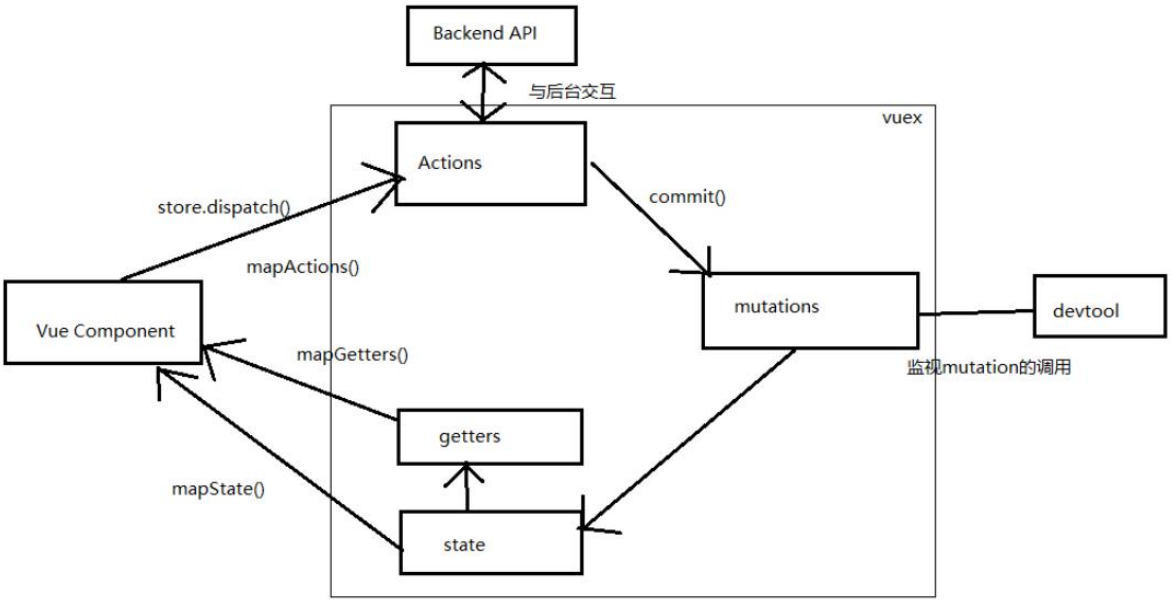
```
body {
  background: #fff;
}

.btn {
  display: inline-block;
  padding: 4px 12px;
  margin-bottom: 0;
  font-size: 14px;
```

```
  line-height: 20px;
  text-align: center;
  vertical-align: middle;
  cursor: pointer;
  box-shadow: inset 0 1px 0 rgba(255, 255, 255, 0.2), 0 1px 2px rgba(0, 0, 0, 0.05);
  border-radius: 4px;
}

.btn-danger {
  color: #fff;
  background-color: #da4f49;
  border: 1px solid #bd362f;
}

.btn-danger:hover {
  color: #fff;
  background-color: #bd362f;
}

.btn:focus {
  outline: none;
}
```

6.3.13. main.js

```
import Vue from 'vue'
import App from './components/app'
import store from './store'

import './base.css'

/* eslint-disable no-new */
new Vue({
  el: '#app',
  render: h => h(App),
  store
})
```

6.5. vuex 结构分析



第 7 章：vue 源码分析

7.1. 说明

1) 分析 vue 作为一个 MVVM 框架的基本实现原理

   数据代理

   模板解析

   数据绑定

2) 不直接看 vue.js 的源码

3) 剖析 github 上某基友仿 vue 实现的 mvvm 库

4) 地址: https://github.com/DMQ/mvvm

7.2. 准备知识

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 是否是 obj 自身的属性

7.3. 数据代理

1) 数据代理: 通过一个对象代理对另一个对象(在前一个对象内部)中属性的操作(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

7.4. 模板解析

7.4.1. 模板解析的基本流程

1) 将 el 的所有子节点取出, 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

7.4.2. 模板解析(1): 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

7.4.3. 模板解析(2): 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

7.4.4. 模板解析(3): 一般指令解析

1) 得到指令名和指令值(表达式) text/html/class msg/myClass
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

7.5. 数据绑定

7.5.1. 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

7.5.2. 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: 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

7.5.3. 四个重要对象

1) Observer
   a. 用来对 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 中的每个属性创建对应的 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. complie 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
    depIds //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

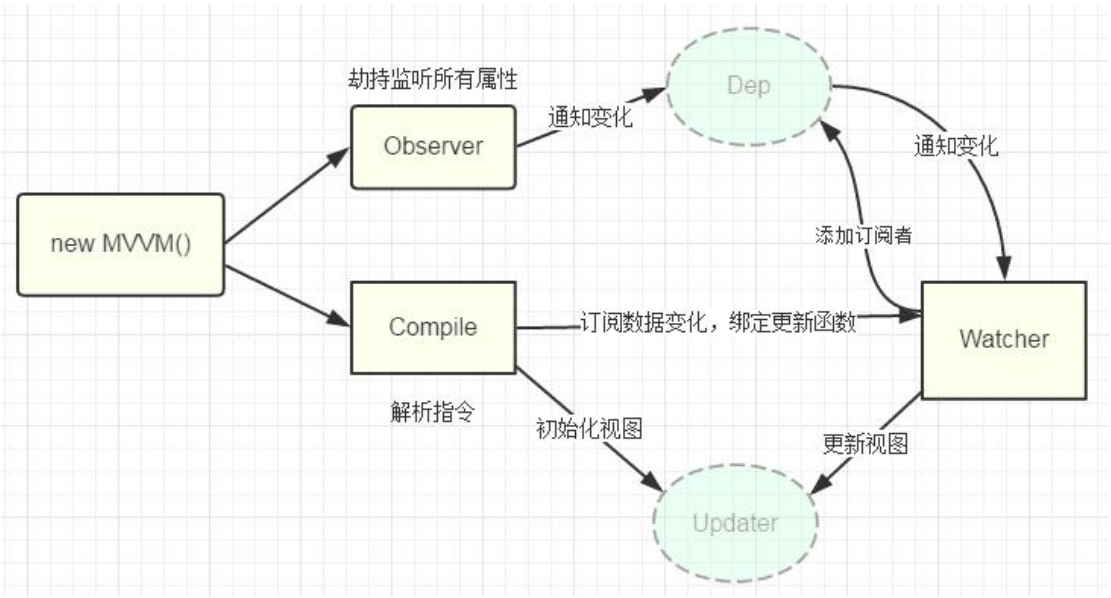
a. data 中的一个属性对应一个 dep，一个 dep 中可能包含多个 watcher(模板中有几个表达式使用到了同一个属性)

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: a.b)

c. 数据绑定使用到 2 个核心技术

* defineProperty()
* 消息订阅与发布

7.6. MVVM 原理图分析



7.7. 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流程:

   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，将最新的值赋值给当前表达式所对应的 data 属性