Vue.js MVVM 基础

渐进式 JavaScript 框架

Content

- Vue 特点
- Vue 实例
- 插值表达式 {{}}
- 指令: v-bind, v-if, v-for, v-model, v-on
- 使用组件构建应用
- Vue 实力属性和方法
- 实例生命周期

Content

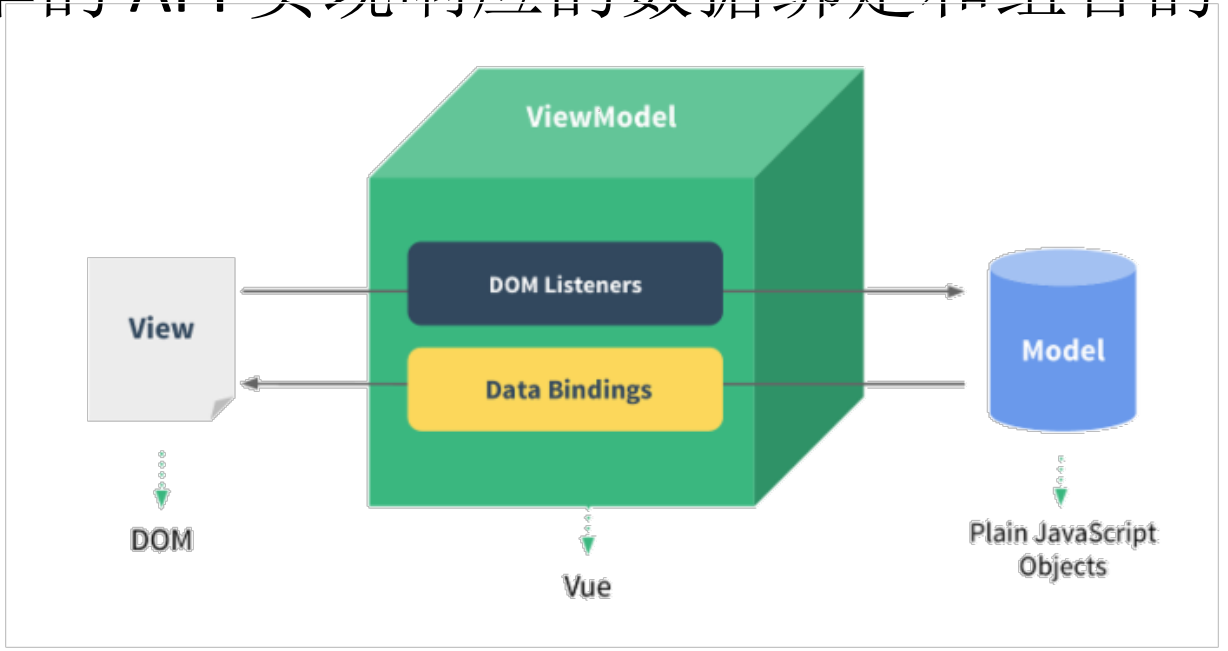
- 模板语法与绑定
- 过滤器 Filter
- 计算属性 computed
- 方法 methods
- 观察 Watchers
- v-bind:class, v-bind:style
- 条件渲染：v-if, v-show
- 列表渲染

Content

- 组件和 v-for
- 数组更新检测：变异方法，非变异方法
- 事件处理器
- $event 特殊变量
- 事件修饰符
- 按键修饰符，内置按键别名，组合按键修饰符
- 表单控件绑定
- 表单绑定修饰符：.lazy, .number, .trim

Vue.js 是什么

- Vue.js（读音 /vju:/, 类似于 view） 是一套构建用户界面的渐进式框架。Vue 采用自底向上增量开发的设计，目标是通过尽可能简单的 API 实现响应的数据绑定和组合的视图组件。



Vue.js 特点

- 文档完整，生态繁荣，工具齐全
- 易用
  - 没有 React, Angular 2 陡峭的学习曲线和复杂度
- 灵活
  - 简单小巧的核心，渐进式技术栈，足以应付任何规模的应用
- 性能
  - 17kb min+gzip 运行大小，超快虚拟 DOM
- 不支持 IE8 及其以下版本

Vue 实例

- 每个 Vue.js 应用都是通过构造函数 **Vue** 创建一个 Vue 的根实例启动的。
- 每个 Vue 实例都会代理其 data 对象里所有的属性

```
var data = { a: 1 }

var vm = new Vue({
    el: '#example',
    data: data
})

vm.a === data.a // -> true
```

插值表达式

- 绑定插入的文本 {{}}

```
<script src="js/vue.min.js"></script>

<div id="app">
    {{ message }}
</div>

<script type="text/javascript">
var app = new Vue({
    el: '#app',
    data: {
        message: 'Hello Vue!'  // 响应式属性
    }
})
</script>
```

指令

- 指令带有前缀 **v-**，以表示它们是 Vue.js 提供的特殊属性
  - v-bind
  - v-if
  - v-for
  - v-model

使用指令绑定 DOM 元素属性

- v-bind 指令:

```
<div id="app-2">
    <span v-bind:title="message">
        鼠标移上来试试!
    </span>
</div>
```

```
<script type="text/javascript">
    var app2 = new Vue({
        el: '#app-2',
        data: {
        message: '页面加载时间为: ' + new Date()
        }
    })
</script>
```

条件与循环

v-if

```
<div id="app-3">
    <p v-if="seen">Now you see me</p>
</div>
```

```
<script type="text/javascript">
    var app3 = new Vue({
        el: '#app-3',
        data: {
            seen: true
        }
    })
</script>
```

条件与循环

- **v-for** 指令可以绑定数据到数组来渲染一个列表：

```
<div id="app-4">
    <ol>
        <li v-for="todo in todos">
        {{ todo.text }}
        </li>
    </ol>
</div>
```

```
<script type="text/javascript">
    var app4 = new Vue({
        el: '#app-4',
        data: {
        todos: [
            { text: 'Learn JavaScript' },
            { text: 'Learn Vue' },
            { text: 'Build something awesome' }
            ]
        }
    })
</script>
```

处理用户输入

- **v-on** 指令绑定一个监听事件调用 Vue 实例中定义的方法

```
<div id="app-5">
    <p>{{ message }}</p>
    <button v-on:click="reverseMsg">
    反转
    </button>
</div>
```

```
var app5 = new Vue({
    el: '#app-5',
    data: {
        message: 'Hello Vue.js!'
    },
    methods: {
        reverseMsg: function() {
            this.message =
            this.message.split('').reverse().joi
            n('')
        }
    }
})
```

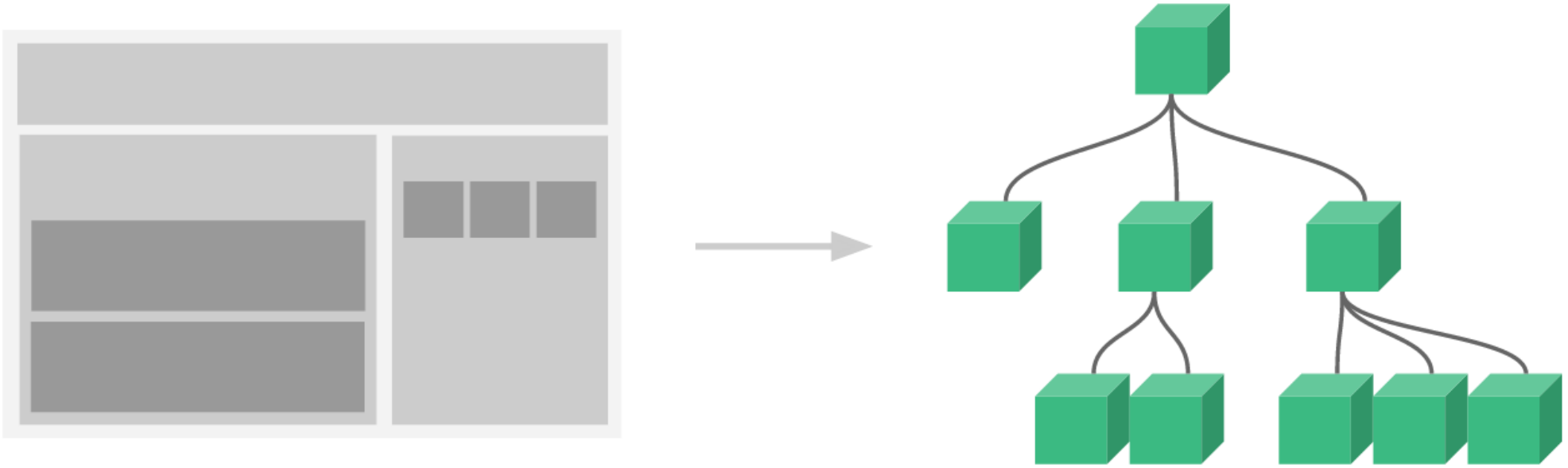
双向数据绑定

- **v-model** 指令在表单输入和应用状态中做双向数据绑定

```
<div id="app-6">
  <p>{{ message }}</p>
  <input v-model="message">
</div>
```

```
var app6 = new Vue({
  el: '#app-6',
  data: {
    message: 'Hello Vue!'
  }
})
```

用组件构建（应用）



用组件构建（应用）

- 一个组件实质上是一个拥有预定义选项的一个 Vue 实例

- 全局组件注册

```
//注册一个全局组件
Vue.component('todo-item', {
    template: '<li>hey</li>' // 组件模板
})
```

- 使用组件

```
<ol id="app-7">
    <todo-item></todo-item>
</ol>
```

用组件构建（应用）

- 带属性的组件

```
<ol id="app-8">
    <todo-item2 v-bind:todo="item" v-for="item in list"></todo-item2>
</ol>

Vue.component('todo-item2', {
    props: ['todo'], // 属性列表
    template: '<li>{{todo.text}}</li>' // 组件模板
})
```

用组件构建（应用）

- 测试

```
new Vue({
    el: '#app-8',
    data: {
        list: [
                { text: "todo1" },
                { text: "todo2" },
                { text: "todo3" }
            ]
    }
})
```

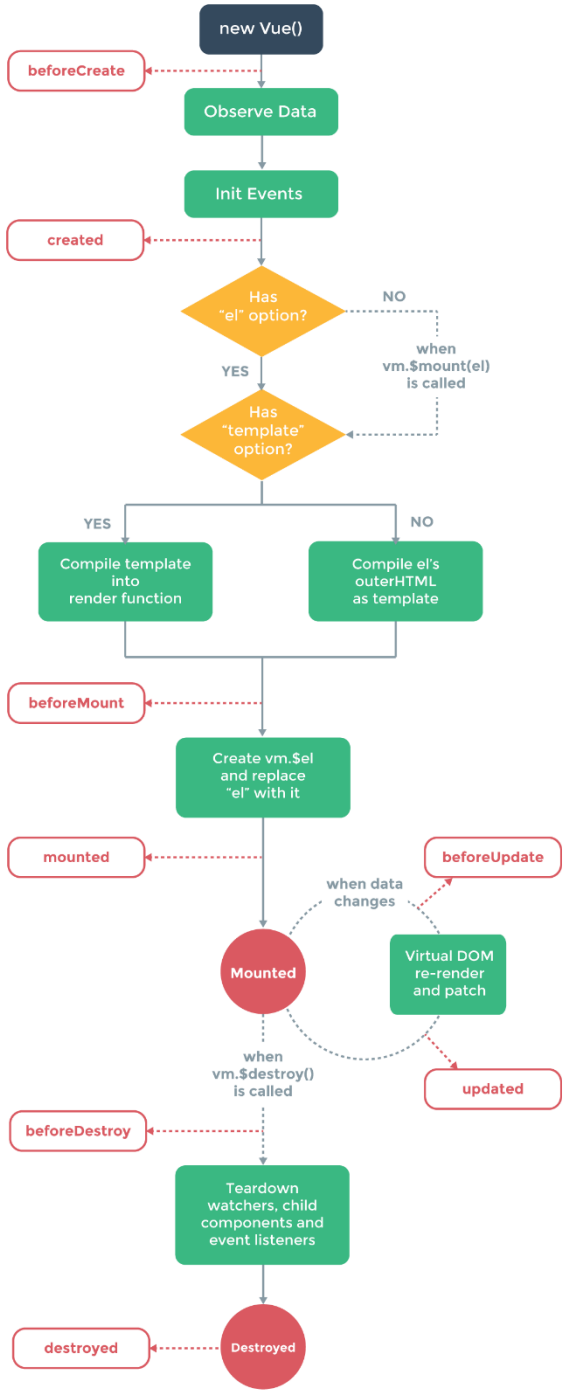
Vue 实例属性和方法

- Vue 实例暴露了一些有 $ 前缀的实例属性与方法

```
var data = { a: 1 }
var vm = new Vue({
    el: '#example',
    data: data
})
vm.$data === data // -> true
vm.$el === document.getElementById('example') // -> true
// $watch 是一个实例方法
vm.$watch('a', function(newVal, oldVal) {
    // 这个回调将在 `vm.a`  改变后调用
    console.info(newVal+" : "+oldVal);
})
vm.a=2;
```

实例生命周期

```
var vm = new Vue({
    el: '#app-3',
    data: {
        message: 'Hello Vue!'
    },
    updated: function() {
        console.info('updated');
    },
    created: function() {
        // `this` 指向 vm 实例
        console.log('message is: ' + this.message)
    },
    destroyed: function() {
        console.info('destroyed');
    }
})
//vm.$destroy();
```



实例生命周期

- beforeCreate
- created
- beforeMount
- mounted
- beforeUpdate
- updated
- beforeDestroy
- destroyed

模板语法

- 插值
  - 插值表达式 {{}}
  - 纯 HTML 输出绑定
  - 属性绑定
  - JavaScript 表达式
- 指令
- Filters
- 缩写

模板语法

- **插值表达式 mustache {{}}**
  - 使用 v-once 指令，能执行一次性地插值，当数据改变时，节点中所有插值处的内容不会更新。

```
<span>Message: {{ msg }}</span>

<span v-once>This will never change: {{ msg }}</span>
```

模板语法

- 纯 HTML 输出绑定
  - 被插入的内容都会被当做 HTML，请只对可信内容使用 HTML 插值，**绝不要**对用户提供的内容插值。防止 XSS 攻击。
  - 不支持插值

```
<div v-html="htmlRaw"></div>

data: {
    message: 'Hello Vue!',
    htmlRaw: '<h1>Hello</h1>'
}
```

模板语法

- 属性绑定
  - Mustache 不能在 HTML 属性中使用，应使用 v-bind 指令：

```
<div v-bind:id="dynamicId"></div>

<button v-bind:disabled="someDynamicCondition">Button</button>
```

JavaScript 表达式

- Vue 支持 JavaScript 表达式
  - 不支持语句定义
  - 不支持流程控制，但可以使用三元表达式

```
{{ number + 1 }}
{{ ok ? 'YES' : 'NO' }}
{{ message.split('').reverse().join('') }}
<div v-bind:id="'list-' + id"></div>
```

指令 Directives

- 指令（Directives）是带有 v- 前缀的特殊属性。

```
<!-- 普通指令 -->
<p v-if="seen">Now you see me</p>

<!-- 带参数的指令: -->

<!--属性-->
<a v-bind:href="url"></a>
<!-- 事件 -->
<a v-on:click="doSomething">

<!-- 修饰符.-->
<form v-on:submit.prevent="onSubmit"></form>
```

过滤器 Filter

- 过滤器只能在 mustache 绑定和 v-bind 表达式中使用

```
{{ message | capitalize }}
```

```
filters: {
    capitalize: function(value) {
        if(!value) return ''
        value = value.toString()
        return value.charAt(0).toUpperCase() + value.slice(1)
    }
},
```

过滤器 Filter

- 过滤器可以串联

```
{{ message | filterA | filterB }}
```

- 滤器可以接受参数
  - 'arg1' 将传给过滤器作为第二个参数， arg2 表达式的值将被求值然后传给过滤器作为第三个参数

```
{{ message | filterA('arg1', arg2) }}
```

指令缩写

- **v-bind:**xxxx ==> **:**xxxx
- **v-on:**xxxx ==> **@**xxxx

计算属性

- 计算属性可以避免在模板中放入太多的处理逻辑，将一些运算在 JS 中计算好后直接返回
  - 计算属性会自动随着依赖属性的变化而变化

```
{{ message }}
{{ reversedMessage }}
```

```
data: {
    message: 'hello Vue!',
},
computed: {
    reversedMessage: function() {
        return this.message.split('').reverse().join('')
    }
},
```

计算属性 VS method

- method 可以实现同样功能，但没有缓存，每次调用都需要重复执行函数。
- 而 computed 计算属性**基于依赖缓存**，仅在依赖的属性变化时才重新执行函数。

```
methods: {
    reverseMessage: function() {
        return this.message.split('').reverse().join('')
    }
},
```

计算属性 setter

- 计算属性支持 setter
  - 在 vm.message2='xxx'; 时可以更新普通属性

```
message2: {
// getter
    get: function () {
      return this.message.toUpperCase();
    },
    // setter
    set: function (newValue) {
      this.message=newValue.toLowerCase();
    }
}
```

观察 Watchers

- 提供属性监控，当属性更改时调用

```
watch: {
    // 如果 message 发生改变，这个函数就会运行
    message: function(newVal) {
    console.info("update: " + newVal);
    }
},
```

Class 绑定

- 数据绑定一个常见需求是操作元素的 class 列表和它的内联样式。
- 在 v-bind 用于 **class** 和 **style** 时， Vue.js 专门增强了它。表达式的结果类型除了字符串之外，还可以是对象或数组。

Class 绑定

- **v-bind:class**
  - 元素上已经存在的类不会被覆盖
- **对象绑定语法**

```
<!-- isActive 为真则应用 active 类样式-->
<div class="static"
     v-bind:class="{ active: isActive, 'text-danger': hasError }">
</div>
```

Class 绑定

- **数组绑定语法**

```
<div v-bind:class="[activeClass, errorClass]">
<div v-bind:class="[{ active: isActive }, errorClass]">
```

```
data: {
  active: true,
  activeClass: 'active',
  errorClass: 'text-danger'
}
```

绑定内联样式

- **v-bind:style** 语法是一个 JavaScript 对象。
  - CSS 属性名可以用**驼峰式（camelCase）**或**短横分隔命名（kebab-case）**
  - 会自动为需要特定前缀的 CSS 属性添加前缀

```
<div v-bind:style="{ color: activeColor, fontSize: fontSize + 'px' }"></div>
```

条件渲染

- v-if, v-else-if, v-else

```
<div v-if="type === 'A'">
A
</div>
<div v-else-if="type === 'B'">
B
</div>
<div v-else-if="type === 'C'">
C
</div>
<div v-else>
Not A/B/C
</div>
```

条件渲染

- <template> 包装元素可以包含多个标签，最终的渲染结果不会包含 template

```
<template v-if="ok">
    <h1>Title</h1>
    <p>Paragraph 1</p>
    <p>Paragraph 2</p>
</template>
```

使用-key-控制元素的可重用

- Vue 尝试尽可能高效的渲染元素，通常会复用已有元素而不是从头开始渲染。
  - 在代码中切换 loginType 不会删除用户已经输入的内容，两个模版由于使用了相同的元素，<input> 会被复用，仅仅是替换了 placeholder。

```
<template v-if="loginType === 'username'">
    <label>Username</label>
    <input placeholder="Enter your username">
</template>
<template v-else="">
    <label>Email</label>
    <input placeholder="Enter your email address2">
</template>
```

使用-key-控制元素的可重用

- 设置每次切换时重新渲染，添加一个属性 key ， key 必须带有唯一的值

```
<template v-if="loginType === 'username'">
    <label>Username</label>
    <input placeholder="Enter your username" key="username-input">
    </template>
    <template v-else>
    <label>Email</label>
    <input placeholder="Enter your email address" key="email-input">
    </template>
</div>
```

v-show

- **v-if**
  - 生成或不生成元素，在切换当中适当地销毁与重建条件块内的事件监听器和子组件。
  - **惰性的：**如果在初始渲染时条件为假，则什么也不做，在条件第一次变为真时才开始局部编译（编译会被缓存起来）。
  - **有更高的切换消耗。适合运行时条件不大可能改变。**
- **v-show**
  - 元素始终被编译并保留，只是简单地基于 CSS 切换显示或隐藏元素。
  - **有更高的初始渲染消耗。适合频繁切换。**

列表渲染

- **v-for** 指令根据一组数组的选项列表进行渲染
  - 需要以 **item in items** 形式的特殊语法
  - items 是源数据数组或对象并且 item 是数组元素迭代的别名
  - 支持一个可选的第二个参数为当前项的索引或对象的 key
  - 也可以用 **of** 替代 **in** 作为分隔符

列表渲染

- v-for

```
<ul id="example-1">
    <li v-for="item in items">
        {{ item.message }}
    </li>
</ul>

<ul id="example-2">
    <li v-for="(item, index) in items">
        {{ parentMessage }} - {{ index }} - {{ item.message }}
    </li>
</ul>

<div v-for="item of items"></div>
```

列表渲染

- 可以用带有 v-for 的 <template> 标签来渲染多个元素块

```
<ul>
    <template v-for="item in items">
        <li>{{ item.msg }}</li>
        <li class="divider"></li>
    </template>
</ul>
```

列表渲染

- 可以用 v-for 通过一个对象的属性来迭代。

```
<ul id="repeat-object" class="demo">
    <li v-for="value in object">
        {{ value }}
    </li>
</ul>

<div v-for="(value, key) in object">
    {{ key }} : {{ value }}
</div>

<div v-for="(value, key, index) in object">
    {{ index }}. {{ key }} : {{ value }}
</div>
```

列表渲染

- v-for 也可以取整数。在这种情况下，它将重复多次模板。

```
<div>
    <span v-for="n in 10">{{ n }}</span>
</div>
```

组件和 v-for

- 在自定义组件里，你可以像任何普通元素一样用 v-for 。然而他不能自动传递数据到组件里，因为组件有自己独立的作用域。
- 为了传递迭代数据到组件里，要用 props 声明接收数据的组件属性，并通过 v-bind 绑定 v-for 的数据到组件：

```
<my-component
v-for="(item, index) in items"
v-bind:item="item"
v-bind:index="index" >

</my-component>
```

```
Vue.component('todo-item2', {
    props: ['item', 'index'], // 属性列表
    template: '<li>{{item}}-{{index}}</li>'
})
```

组件和 v-for

- 为了让 Vue 能跟踪每个节点的身份，从而重用和重新排序现有元素，你需要为每项提供一个唯一 key 属性（相当于 Vue 1.x 的 track-by ），它的工作方式类似于一个属性，所以需要用 v-bind 来绑定动态值：

```
<div v-for="item in items" :key="item.id">
    <!-- 内容 -->
</div>
```

数组更新检测

- **变异方法**(mutation method)
  - 会改变被这些方法调用的原始数组，能触发视图更新
  - push()
  - pop()
  - shift()
  - unshift()
  - splice()
  - sort()
  - reverse()

数组更新检测

- **非变异(non-mutating method)方法**
  - 不会改变原始数组，但总是返回一个新数组。
  - 当使用非变异方法时，可以用新数组替换旧数组。
  - filter()
  - concat()
  - slice()

数组更新检测

- 不能检测以下变动的数组：

- 利用索引直接设置一个项

```
vm.items[indexOfItem] = newValue
```

- 修改数组的长度

```
vm.items.length = newLength
```

数组更新检测

- vm.items[indexOfItem] = newValue 替代方案

```
// Vue.set
Vue.set(vm.items, indexOfItem, newValue)

// Array.prototype.splice`
vm.items.splice(indexOfItem, 1, newValue)
```

数组更新检测

- vm.items.length = newLength 替代方案

```
vm.items.splice(newLength)
```

显示过滤-排序结果

- 使用 computed 计算属性或 method 方法，可以显示一个数组的过滤或排序副本，而不实际改变或重置原始数据。

```
<li v-for="n in evenNumbers">{{ n }}</li>
```

```
data: {
  numbers: [ 1, 2, 3, 4, 5 ]
},
computed: {
  evenNumbers: function () {
    return this.numbers.filter(function (number) {
      return number % 2 === 0
    })
  }
}
```

```
<li v-for="n in even(numbers)">{{ n }}</li>
```

```
data: {
  numbers: [ 1, 2, 3, 4, 5 ]
},
methods: {
  even: function (numbers) {
    return numbers.filter(function (number) {
      return number % 2 === 0
    })
  }
}
```

事件处理器

- v-on 指令监听 DOM 事件来触发一些 JavaScript 代码。

```
<button v-on:click="counter += 1">增加 1</button>
```

事件处理器

- 事件可以绑定或调用 methods 中的方法
  - 如果调用时没有传入参数，方法默认有一个 event 参数
  - 如果需要在有参数的情况下获得 DOM 事件对象，可以使用 $event 变量

```
<button @click="greet">
Greet
</button>
<button @click="say('hi')">
Say hi
</button>
<button @click="say('what')">
Say what
</button>
```

```
methods: {
    greet: function(event) {
        // `this` 在方法里指当前 Vue 实例
        alert('Hello ' + this.name + '!')
        // `event` 是原生 DOM 事件
        alert(event.target.tagName)
    },
    say: function(message) {
        alert(message)
    }
}
```

事件处理器

- 使用 $event 特殊变量

```
<button v-on:click="warn('Cannot be submitted.', $event)">Submit</button>

methods: {
  warn: function (message, event) {
    // 现在我们可以访问原生事件对象
    if (event) event.preventDefault()
    alert(message)
  }
}
```

事件修饰符

- Vue 建议 methods 只有纯粹的数据逻辑，而不去处理 DOM 事件细节。所以，Vue.js 为 v-on 提供了 事件修饰符。通过由**点(.)**表示的指令后缀来调用修饰符。
  - .stop
  - .prevent
  - .capture
  - .self
  - .once

事件修饰符

```
<!-- 阻止单击事件冒泡 -->
<a v-on:click.stop="doThis"></a>
<!-- 阻止提交动作 -->
<form v-on:submit.prevent="onSubmit"></form>
<!-- 串联：阻止冒泡和动作 -->
<a v-on:click.stop.prevent="doThat"></a>
<!-- 只有修饰符 -->
<form v-on:submit.prevent></form>
<!-- 添加事件侦听器时使用事件捕获模式 -->
<div v-on:click.capture="doThis">...</div>
<!-- 只当事件在该元素本身（而不是子元素）触发时触发回调 -->
<div v-on:click.self="doThat">...</div>
<!-- 单击事件仅触发一次 -->
<a v-on:click.once="doThis"></a>
```

按键修饰符

- Vue 允许为 v-on 在监听键盘事件时添加按键修饰符：
  - 可以使用按键数值或按键别名

```
<!-- 只有在 keyCode 是 13 时调用 vm.submit() -->
<input v-on:keyup.13="submit">

<!-- 使用按键别名 -->
<input @keyup.enter="submit">
```

内置按键别名

- .enter
- .tab
- .delete (捕获 “删除” 和 “退格” 键)
- .esc
- .space
- .up
- .down
- .left
- .right

自定义按键别名

- 可以通过全局 **config.keyCodes** 对象自定义按键修饰符别名：

```
// 可以使用 v-on:keyup.f1
Vue.config.keyCodes.f1 = 112
```

组合按键修饰符

- .ctrl
- .alt
- .shift
- .meta（Windows, Command）

```
<!-- Alt + C -->
<input @keyup.alt.67="clear">
<!-- Ctrl + Click -->
<div @click.ctrl="doSomething">Do something</div>
```

表单控件绑定

- **v-model** 指令在表单控件元素上创建双向数据绑定。

```
<!-- 文本框和文本域-->
<input v-model="message" placeholder="edit me">
<textarea v-model="message" placeholder="add multiple lines"></textarea>
<p>Message is: {{ message }}</p>

<!-- 复选框 -->
<input type="checkbox" id="checkbox" v-model="checked">
<label for="checkbox">{{ checked }}</label>

<!-- 单选按钮 -->
<input type="radio" value="One" v-model="picked"> One
<input type="radio" value="Two" v-model="picked"> Two
<span>Picked: {{ picked }}</span>
```

表单控件绑定

- 多个复选框，绑定到同一个数组：

```
<input type="checkbox" id="jack" value="Jack" v-model="checkedNames">
<label for="jack">Jack</label>
<input type="checkbox" id="john" value="John" v-model="checkedNames">
<label for="john">John</label>
<input type="checkbox" id="mike" value="Mike" v-model="checkedNames">
<label for="mike">Mike</label>
<br>
<span>Checked names: {{ checkedNames }}</span>
```

表单控件绑定

- 下拉菜单 v-for 动态渲染和绑定

```
<select v-model="selected">
  <option v-for="option in options" v-
bind:value="option.value">
    {{ option.text }}
  </option>
</select>
<span>Selected: {{ selected }}</span>
```

```
selected: 'A',
options: [
    { text: 'One', value: 'A' },
    { text: 'Two', value: 'B' },
    { text: 'Three', value: 'C' }
],
```

表单控件绑定

- v-bind 可以绑定 value 到 Vue 实例的一个动态属性上，并且这个属性的值可以不是字符串。

```
<input
  type="checkbox"
  v-model="toggle"
  v-bind:true-value="a"
  v-bind:false-value="b"
>
```

```
// 当选中时
vm.toggle === vm.a
// 当没有选中时
vm.toggle === vm.b
```

表单绑定修饰符

- **.lazy**

在 change 事件中进行数据同步。

```
<!-- 在 "change" 而不是 "input" 事件中更新 -->
<input v-model.lazy="message" >
{{message}}
```

表单绑定修饰符

- **.number**

自动将用户的输入值转为 Number 类型（如果原值的转换结果为 NaN 则返回原值）

```
<input v-model.number="message" >
{{message+1}}
```

表单绑定修饰符

- **.trim**

自动过滤用户输入的首尾空格

```
<input v-model.trim="msg">
```

End

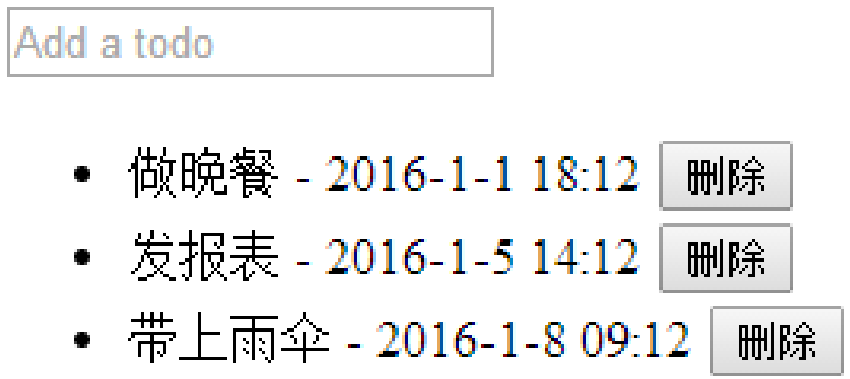
Thanks!

任务

练习

任务一

- 使用自定义组件实现如下 todo list
  - 输入内容，回车，自动添加新 todo，文本框清空
  - 点击删除，删除对应 todo

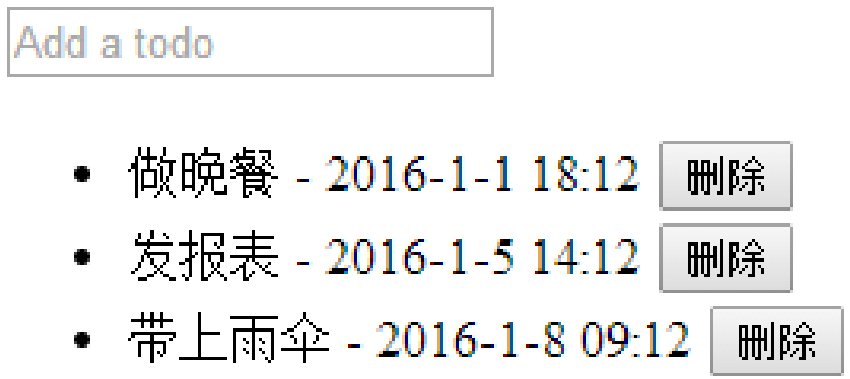


任务一

- 使用监听事件，实现点击按钮数量增加

任务二

- 使用自定义组件实现如下 todo list
  - 输入内容，回车，自动添加新 todo，文本框清空
  - 点击删除，删除对应 todo



任务三

- 实现如下表格

Create New Person

Name: S

Age: 0

Sex: Male

Create

Name	Age	Sex	Delete
Jack	30	Male	Delete
Bill	26	Male	Delete
Tracy	22	Female	Delete
Chris	36	Male	Delete
Steve	18	Male	Delete

任务四

- GitHub 提交
  - 点击单选按钮切换显示