Table of Contents

概述

Vue.js快速入门

第 1 章 Vue.js简介

近几年，互联网前端行业发展得依旧迅猛，涌现出了很多优秀的框架，同时这些框架也正在逐渐改变我们传统的前端开发方式。Google的AngularJS、Facebook的ReactJS，这些前端MVC（MVVM）框架的出现和组件化开发的普及和规范化，既改变了原有的开发思维和方式，也使得前端开发者加快脚步，更新自己的知识结构。2014年2月，原Google员工尤雨溪公开发布了自己的前端库——Vue.js，时至今日，Vue.js在GitHub上已经收获超过30000star，而且也有越来越多的开发者在实际的生产环境中运用它。

本书主要以Vue.js 1.0.26版本为基准进行说明，Vue.js 2.0版本与之不同的地方，会在对应章节中说明。

1.1 Vue.js是什么

Vue.js被定义成一个用来开发Web界面的前端库，是个非常轻量级的工具。Vue.js本身具有响应式编程和组件化的特点。Vue.js一直以轻量级，易上手被称道。MVVM的开发模式也使前端从原先的DOM操作中解放出来，我们不再需要在维护视图和数据的统一上花大量的时间，只需要关注于data的变化，代码变得更加容易维护。

1. 第一个特性是数据绑定
2. 第二个特性是组件化

1.2 Vue.js的Hello world

引入vue的方法

1. script标签引入
2. npm install vue

使用

HTML文件中的内容为

```
<div id="app">
  <h1></h1>
</div>
```

应用的js如下：

```
var vm = new Vue({
  el : '#app',
  data: {
    message : 'Hello world, I am Vue.js'
  }
});
```

输出结果为：Hello world, I am Vue.js

第 2 章 基础特性

- 模板渲染
- 事件绑定
- 处理用户交互（输入信息或鼠标操作）

2.1 实列及选项

Vue.js的使用都是通过构造函数**Vue({option})**创建一个**Vue**的实例：

```
var vm = new Vue({})
```

一个**Vue**实例相当于一个**MVVM**模式中的**ViewModel**

在实例化的时候，我们可以传入一个选项对象，包含数据、模板、挂载元素、方法、生命周期钩子等选项。

2.1.1 模板

- el：类型为字符串，DOM元素或函数。 一般来说我们会使用css选择符，或者原生的DOM元素。 `el:#app` 初始项中指定了el，实例将立即进入编译过程。
- template：类型为字符串。 默认会将template值替换挂载元素（即el值对应的元素），并合并挂载元素和模板根节点的属性（如果属性具有唯一性，类似id，则以模板根节点为准） ~~如果replace为false，模板template的值将插入挂载元素内。~~ 通过template插入模板的时候，挂载元素的内容都将被互联，除非使用slot进行分发

  `<scrip id="tpl" type="x-template">`

Vue.js 2.0 中废除了**replace**这个参数，并且强制要求每一个**Vue.js**实例需要有一个根元素

2.1.2 数据

通过**data**属性定义数据

传入**data**的是一个对象，**Vue**实例会代理起**data**对象里的所有属性，而不会对传入的对象进行深拷贝

可以引用**Vue**实例**vm**中的**$data**来获取声明的数据

```
    var data = { a: 1 }
    var vm = new Vue({
     data: data
    })
    vm.$data === data // -> true
    vm.a === data.a // -> true
    // 设置属性也会影响到原始数据
    vm.a = 2
    data.a // -> 2
    // 反之亦然
    data.a = 3
    vm.a // -> 3
```

只有初始化时传入的对象才是响应式的

如果需要在实例化之后加入响应式变量，需要调用实例方法**$set**

```
vm.$set('b', 2);
```

不过**Vue.js**并不推荐这么做，这样会抛出一个异常。我们应尽量在初始化的时候，把所有的变量都设定好，如果没有值，也可以用**undefined**或**null**占位

组件类型的实例可以通过**props**获取数据，同**data**一样，也需要在初始化时预设好

我们也可以在上述组件类型实例中同时使用**data**，但有两个地方需要注意：

> 1. data的值必须是一个函数，并且返回值是原始对象。如果传给组件的data是一个原始对象的话，则在建立多个组件实例时它们就会共用这个data对象，修改其中一个组件实例的数据就会影响到其他组件实例的数据
> 2. data中的属性和props中的不能重名。这两者均会抛出异常

所以正确的使用方法如下：

```
var MyComponent = Vue.component('my-component', {
   props : ['title', 'content'],
   data : function() {
     return {
       desc : '123'
     }
   },
   template : '<div> \
      <h1></h1> \
      <p></p> \
      <p></p> \
    </div>'
})
```

2.1.3 方法

通过选项属性**methods**对象来定义方法，并且使用**v-on**指令来监听**DOM**事件

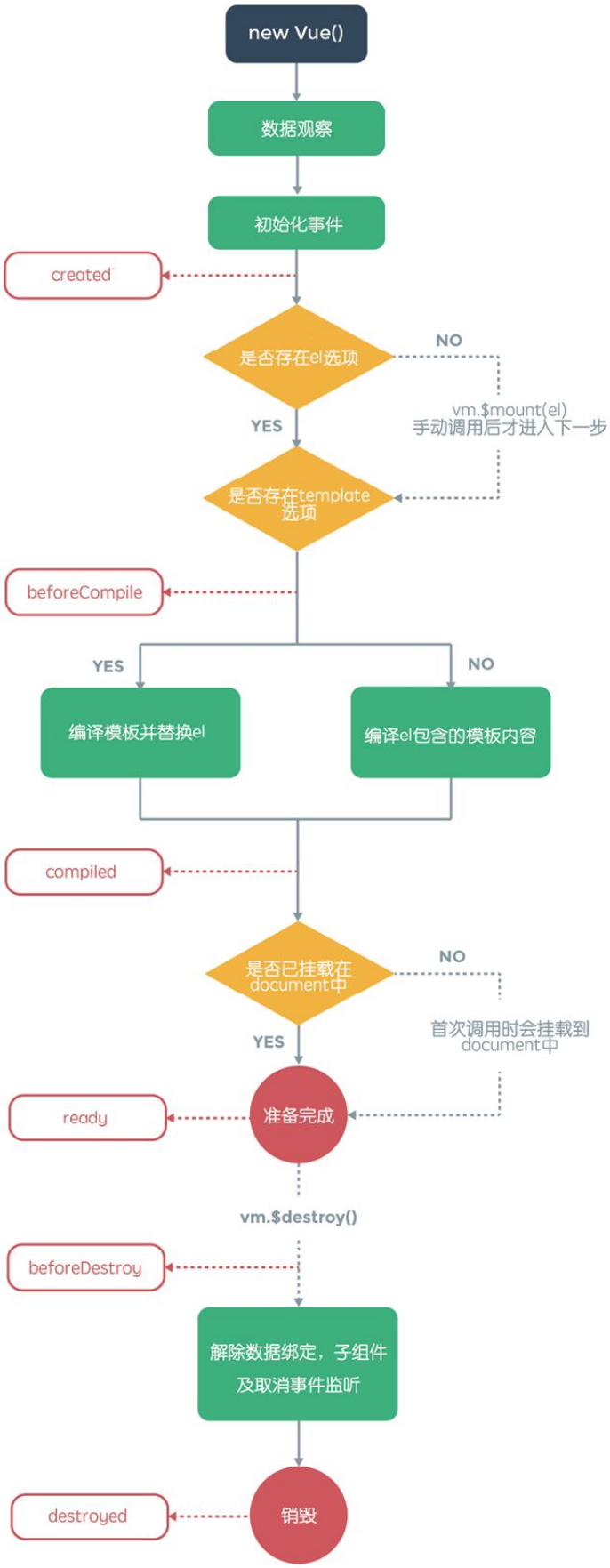
~~Vue.js实例也支持自定义事件，可以在初始化时传入events对象，通过实例的$emit方法进行触发~~

Vue.js 2.0 中废弃了**events**选项属性，不再支持事件广播这类特性，推荐直接使用**Vue**实例的全局方法**$on()/$emit()**，或者使用插件**Vuex**来处理。

2.1.4 生命周期

Vue.js实例在创建时有一系列的初始化步骤

在此过程中，我们可以通过一些定义好的生命周期钩子函数来运行业务逻辑


new Vue()
数据观察
初始化事件
created
是否存在el选项
NO
vm.$mount(el)
手动调用后才进入下一步
YES
是否存在template选项
beforeCompile
YES
NO
编译模板并替换el
编译el包含的模板内容
compiled
是否已挂载在document中
NO
首次调用时会挂载到document中
YES
ready
准备完成
vm.$destroy()
beforeDestroy
解除数据绑定，子组件及取消事件监听
destroyed
销毁

- ~~init:~~ 在实例开始初始化时同步调用。此时数据观测、事件等都尚未初始化。2.0中更名为**beforeCreate**
- **created**：在实例创建之后调用。此时已完成数据绑定、事件方法，但尚未开始DOM编译，即未挂载到document中。
- ~~beforeCompile:~~ 在DOM编译前调用。2.0废弃了该方法，推荐使用created。
- **beforeMount**: 2.0新增的生命周期钩子，在mounted之前运行。
- ~~compiled:~~ 在编译结束时调用。此时所有指令已生效，数据变化已能触发DOM更新，但不保证$el已插入文档。2.0中更名为**mounted**。
- ~~ready:~~ 在编译结束和$el第一次插入文档之后调用。2.0废弃了该方法，推荐使用mounted。这个变化其实已经改变了ready这个生命周期状态，相当于取消了在$el首次插入文档后的钩子函数。
- ~~attached:~~ 在vm.$el插入DOM时调用，ready会在第一次attached后调用。2.0废弃了该方法，推荐在其他钩子中自定义方法检查是否已挂载。
- ~~detached:~~ 同attached类似，该钩子在vm.$el从DOM删除时调用，而且必须是指令或实例方法。2.0中同样废弃了该方法。
- **beforeDestroy**: 在开始销毁实例时调用，此刻实例仍然有效。
- **destroyed**: 在实例被销毁之后调用。此时所有绑定和实例指令都已经解绑，子实例也被销毁。
- **beforeUpdate**: 2.0新增的生命周期钩子，在实例挂载之后，再次更新实例（例如更新data）时会调用该方法，此时尚未更新DOM结构。
- **updated**:2.0新增的生命周期钩子，在实例挂载之后，再次更新实例并更新完DOM结构后调用。
- **activated**：2.0新增的生命周期钩子，需要配合动态组件keep-live属性使用。在动态组件初始化渲染的过程中调用该方法。
- **deactivated**：2.0新增的生命周期钩子，需要配合动态组件keep-live属性使用。在动态组件移出的过程中调用该方法。

2.2 数据绑定

Vue.js的核心是一个响应式的数据绑定系统，建立绑定后，DOM将和数据保持同步，这样就无需手动维护DOM，使代码能够更加简洁易懂、提升效率。

2.2.1 数据绑定语法

2.2.2 计算属性

computed

除了在模板中绑定表达式或者利用过滤器外，Vue.js还提供了计算属性这种方法，避免在模板中加入过重的业务逻辑，保证模板的结构清晰和可维护性。

1. 基础例子

```
var vm = new Vue({
      el : '#app,
      data: {
       firstName : 'Gavin',
       lastName: 'CLY'
      }
      computed : {
       fullName : function() {
        // this 指向vm实例
        return this.firstName + ' ' + this.lastName
       }
      }
    });
```

1. setter

```
    var vm = new Vue({
      el : '#el',
      data: {
       cents : 100,
      }
      computed : {
       price : {
         set : function(newValue) {
          this.cents = newValue * 100;
         },
         get : function() {
          return (this.cents / 100).toFixed(2);
         }
       }
      }
    });
```

在使用Vue.js的计算属性后，我们可以将vm.cents 设置为后端所存的数据，计算属性price为前端展示和更新的数据。

```
<p>&yen;</p>  // ¥1.00
```

此时更改vm.price = 2，vm.cents会被更新为200，在传递给后端时无需再手动转化一遍数据。

2.2.3 表单控件

v-model的指令对表单元素进行双向数据绑定

1. Text
   输入框示例

```
    <input type="text" v-model="message" />
    <span>Your input is : </span>
```

1. Radio
   单选框示例

```
<label><input type="radio" value="male" v-model="gender">男</lable>
<label><input type="radio" value="female" v-model="gender">女</lable>
<p></p>
```

1. Checkbox
   Checkbox分两种情况：单个勾选框和多个勾选框。 单个勾选框，v-model即为布尔值，此时input的value并不影响v-model的值。

```
    <input type="checkbox" v-modal="checked">
    <span>checked: </span>
```

多个勾选框，v-model使用相同的属性名称，且属性为数组。

```
    <label><input type="checkbox" value="1" v-model="multiChecked">1</lable>
    <label><input type="checkbox" value="2" v-model="multiChecked">2</lable>
    <label><input type="checkbox" value="3" v-model="multiChecked">3</lable>
    <p>MultiChecked: \{\{ multiChecked.join('|') \}\}</p>
```

1. Select
   同Checkbox元素一样，Select也分单选和多选两种，多选的时候也需要绑定到一个数组。 单选：

```
    <select v-model="selected">
      <option selected>A</option>
      <option>B</option>
      <option>C</option>
    </select>
    <span>Selected: </span>
```

多选：

```
    <select v-model="multiSelected" multiple>
      <option selected>A</option>
      <option>B</option>
      <option>C</option>
    </select>
    <span>MultiSelected: \{\{ multiSelected.join('|') \}\}</span>
```

1. 绑定value
   表单控件的值同样可以绑定在Vue实例的动态属性上，用v-bind实现

   i. Checkbox

```
<input type="checkbox" v-model="checked" v-bind:true-value="a" v-bind:false-value="b">
选中：vm.checked == vm.a    // -> true
未选中：vm.checked == vm.b  // -> true
```

```
2. Radio
```

```
    <input type="radio" v-model="checked", v-bind:value="a">
    选中：vm.checked == vm.a  // -> true
```

```
3. Select Options
```

```
    <select v-model="selected">
        <!-- 对象字面量 -->
        <option v-bind:value="{ number: 123 }">123</option>
    </select>
    选中：
    typeof vm.selected //  -> 'object'
    vm.selected.number // -> 123
```

1. 参数特性
   Vue.js为表单控件提供了一些参数，方便处理某些常规操作。
   i. ~~lazy~~ 默认情况下，v-model 在input 事件中同步输入框值与数据，加lazy属性后从会改到在 change 事件中同步。

   ```
   <input v-model="query" lazy />
   ```

   ii. ~~number~~ 会自动将用户输入转为Number类型，如果原值转换结果为NaN则返回原值。

   ```
   <input v-model="age" number/>
   ```

   iii. ~~debounce~~ debounce为设置的最小延时，单位为ms，即为单位时间内仅执行一次数据更新。该参数往往应用在高耗操作上，例如在更新时发出ajax请求返回提示信息。

   ```
   <input v-model="query" debounce="500" />
   ```

Vue.js 2.0中取消了lazy和number作为参数，用修饰符（modifier）来代替：

新增了trim修饰符，去掉输入值首尾空格：

```
<input v-model.trim="name" />
```

去除了debounce这个参数，原因是无法监测到输入新数据，但尚未同步到vm实例属性时这个状态。

2.2.4 Class与Style绑定

在开发过程中，我们经常会遇到动态添加类名或直接修改内联样式（例如tab切换）。class和style都是DOM元素的attribute，我们当然可以直接使用v-bind对这两个属性进行数据绑定，例如 `<p v-bind:style='style'><p>` ，然后通过修改vm.style的值对元素样式进行修改。但这样未免过于繁琐而且容易出错，所以Vue.js为这两个属性单独做了增强处理，表达式的结果类型除了字符串之外，还可以是对象和数组。本小节就会对这两个属性具体的用法进行说明。

1. Class绑定

2. 对象语法：
   v-bind:class接受参数是一个对象，而且可以与普通的class属性共存。

   ```
   <div class="tab" v-bind:calss="{'active' : active , 'unactive' : !active}">
   </div>
   ```

   vm实例中需要包含

   ```
   data : {
       active : true
     }
   ```

   渲染结果为： `<div class="tab active"></div>`

3. 数组语法：
   v-bind:class也接受数组作为参数。

   `<div v-bind:class="[classA, classB]"></div>`

   ```
   data : {
       classA : 'class-a',
       classB : 'class-b'
     }
   ```

   渲染结果为： `<div class="class-a class-b"></div>` 。

   也可以使用三元表达式切换数组中的class，

   `<div v-bind:class="[classA, isB ? classB : '']"></div>` 。

   如果vm.isB = false, 则渲染结果为
   `<div v-bind:class="class-a"></div>` 。

4. 内联样式绑定

   style属性绑定的数据即为内联样式，同样具有对象和数组两种形式：

   - 对象语法：直接绑定符合样式格式的对象。

     `<div v-bind:style="alertStyle"></div>`

```
data : {
   alertStyle : {
     color : 'red',
     fontSize : '20px'
   }
}
```

除了直接绑定对象外，也可以绑定单个属性或直接使用字符串。

```
<div v-bind:style="{ fontSize : alertStyle.fontSize, color : 'red'}"></div>
```

- 数组语法：v-bind:style 允许将多个样式对象绑定到统一元素上。

```
<div v-bind:style="[ styleObjectA, styleObjectB]" .></div>
```

5. 自动添加前缀

在使用**transform**这类属性时，**v-bind:style**会根据需要自动添加厂商前缀。**:style**在运行时进行前缀探测，如果浏览器版本本身就支持不加前缀的**css**属性，那就不会添加。

2.3 模板渲染

当获取到后端数据后，我们会把它按照一定的规则加载到写好的模板中，输出成在浏览器中显示的HTML，这个过程就称之为渲染。而Vue.js是在前端（即浏览器内）进行的模板渲染。

2.3.1 前后端渲染对比

前端渲染的优点：

- 业务分离，后端只需要提供数据接口，前端在开发时也不需要部署对应的后端环境，通过一些代理服务器工具就能远程获取后端数据进行开发，能够提升开发效率。
- 计算量转移，原本需要后端渲染的任务转移给了前端，减轻了服务器的压力。

后端渲染的优点：

- 对搜索引擎友好。
- 首页加载时间短，后端渲染加载完成后就直接显示HTML，但前端渲染在加载完成后还需要有段js渲染的时间。

2.3.2 条件渲染

Vue.js 提供v-if，v-show，v-else，v-for这几个指令来说明模板和数据间的逻辑关系，这基本就构成了模板引擎的主要部分。

1. v-if/v-else 可以利用v-else来配合v-if使用 v-else必须紧跟v-if，不然该指令不起作用
2. v-show 也可以搭配v-else使用，用法和v-if一致 与v-if不同的是，v-show元素的使用会渲染并保持在DOM中。v-show只是切换元素的css属性display。
3. v-if vs v-show 当v-if和v-show的条件发生变化时，v-if引起了dom操作级别的变化，而v-show仅发生了样式的变化，从切换的角度考虑，v-show消耗的性能要比v-if小。 v-if切换时，Vue.js会有一个局部编译/卸载的过程，因为v-if中的模板也可能包括数据绑定或子组件。v-if会确保条件块在切换当中适当地销毁与中间内部的事件监听器和子组件。而且v-if是惰性的，如果在初始条件为假时，v-if本身什么都不会做，而v-show则仍会进行正常的操作，然后把css样式设置为display:none。 所以，总的来说，v-if有更高的切换消耗而v-show有更高的初始渲染消耗，我们需要根据实际的使用场景来选择合适的指令。

2.3.3 列表渲染

v-for指令主要用于列表渲染，将根据接收到数组重复渲染v-for绑定到的DOM元素及内部的子元素，并且可以通过设置别名的方式，获取数组内数据渲染到节点中。

v-for内置了$index变量，可以在v-for指令内调用，输出当前数组元素的索引。另外，我们也可以自己指定索引的别名

```
<li v-for="(index,item) in items"> - - </li>
```

以下两种情况是无法触发视图更新

1. 通过索引直接修改数组元素 `vm.items[0] = { title : 'title-changed'};`
2. 无法直接修改“修改数组”的长度 `vm.items.length = 0`

对于第一种情况，Vue.js提供了$set方法，在修改数据的同时进行视图更新，可以写成:

`vm.items.$set(0, { title : 'title-changed'}` 或者 `vm.$set('items[0]', { title : 'title- also-changed '})`

如果数组中有唯一标识id,通过trace-by给数组设定唯一标识

v-for除了可以遍历数组外，也可以遍历对象，与$index类似，作用域内可以访问另一内置变量$key，也可以使用（key, value）形式自定义key变量。

v-for还可以接受单个整数，用作循环次数

2.3.4 template标签用法

v-show和v-if指令都包含在一个根元素中，那是否有方式可以将指令作用到多个兄弟DOM元素上？Vue.js提供了template标签，我们可以将指令作用到这个标签上，但最后的渲染结果里不会有它。

```
<template v-if="yes">
 <p>There is first dom</p>
 <p>There is second dom</p>
 <p>There is third dom</p>
</template>
```

template标签也支持使用v-for指令，用来渲染同级的多个兄弟元素。

2.4 事件绑定与监听

v-on指令用来监听DOM事件，通常在模板内直接使用，而不像传统方式在js中获取DOM元素，然后绑定事件

2.4.1 方法及内联语句处理器

通过v-on可以绑定实例选项属性methods中的方法作为事件的处理器，v-on:后参数接受所有的原生事件名称

v-on的缩写形式@ `<button @click='say'>Say</button>`

除了直接绑定methods函数外，v-on也支持内联JavaScript语句，但仅限一个语句。

在直接绑定methods函数和内联JavaScript语句时，都有可能需要获取原生DOM事件对象，以下两种方式都可以获取：

```
<button v-on:click="showEvent">Event</button>
<button v-on:click="showEvent($event)">showEvent</button>
<button v-on:click="showEvent()">showEvent</button> // 这样写获取不到event
```

```
var vm = new Vue({
 el : '#app',
 methods : {
  showEvent : function(event) {
  console.log(event);
  }
 }
});
```

同一元素上也可以通过v-on绑定多个相同事件函数，执行顺序为顺序执行，

```
<div v-on:click="sayFrom('first')" v-on:click="sayFrom('second)">
```

2.4.2 修饰符

Vue.js为指令v-on提供了多个修饰符，方便我们处理一些DOM事件的细节，并且修饰符可以串联使用

- .stop: 等同于调用event. stopPropagation()。
- .prevent: 等同于调用event.preventDefault()。
- .capture: 使用capture模式添加事件监听器。
- .self: 只当事件是从监听元素本身触发时才触发回调。

```
<a v-on:click.stop='doThis'></a>
<form v-on:submit.prevent="onSubmit"></form> // 阻止表单默认提交事件
<form v-on:submit.stop.prevent="onSubmit"></form> // 阻止默认提交事件且阻止冒泡
<form v-on:submit.stop.prevent></form> // 也可以只有修饰符，并不绑定事件
```

除了事件修饰符之外，v-on还提供了按键修饰符，方便我们监听键盘事件中的按键。

```
<input v-on:keyup.13="submit"/> // 监听input的输入，当输入回车时触发Submit函数（回车的
```

Vue.js给一些常用的按键名提供了别称，这样就省去了一些记keyCode的事件。全部按键别名为：enter、tab、delete、esc、space、up、down、left、right。

```
<input v-on:keyup.enter="submit" />
```

Vue.js也允许我们自己定义按键别名

```
Vue.directive('on').keyCodes.f1 = 112; // 即可以使用<input v-on:keyup.f1="help" />
```

Vue.js 2.0 中可以直接在Vue.config.keyCodes里添加自定义按键别名，无需修改v-on指令

```
Vue.config.keyCodes.f1 = 12
```

2.4.3 与传统事件绑定的区别

Vue.js事件处理方法和表达式都严格绑定在当前视图的ViewModel上，所以并不会导致维护困难。

而这么写的好处在于：

1. 无需手动管理事件。ViewModal被销毁时，所有的事件处理器都会自动被删除，让我们从获取DOM绑定事件然后在特定情况下再解绑这样的事情中解脱出来。

1. 解耦。ViewModel代码是纯粹的逻辑代码，和DOM无关，有利于我们写自动化测试用例。

2.5 Vue.extend()

组件化开发也是Vue.js中非常重要的一个特性，我们可以将一个页面看成一个大的根组件，里面包含的元素就是不同的子组件，子组件也可以在不同的根组件里被调用。在上述例子中，可以看到在一个页面中通常会声明一个Vue的实例new Vue({})作为根组件，那么如何生成可被重复使用的子组件呢？Vue.js提供了Vue.extend(options)方法，创建基础Vue构造器的“子类”，参数options对象和直接声明Vue实例参数对象基本一致，

```
var Child = Vue.extend({
 template : '#child',
 // 不同的是，el和data选项需要通过函数返回值赋值，避免多个组件实例共用一个数据
 data : function() {
  return {
   ….
  }
 }
 ….
})
Vue.component('child', Child)  // 全局注册子组件
<child ….></child>  // 子组件在其他组件内的调用方式
```

更多组件的使用方法将会在第6章中进行详细的说明。

第 3 章 指令

指令是Vue.js中一个重要的特性，主要提供了一种机制将数据的变化映射为DOM行为。那什么叫数据的变化映射为DOM行为？前文中阐述过Vue.js是通过数据驱动的，所以我们不会直接去修改DOM结构，不会出现类似于 `$('ul').append('<li>one</li>')` 这样的操作。当数据变化时，指令会依据设定好的操作对DOM进行修改，这样就可以只关注数据的变化，而不用去管理DOM的变化和状态，使得逻辑更加清晰，可维护性更好。 Vue.js本身就提供了大量的内置指令来进行对DOM的操作，我们也可以开发自定义指令

3.1 内置指令

本节主要介绍Vue.js的内置指令。

3.1.1 v-bind

v-bind主要用于动态绑定DOM元素属性（attribute），即元素属性实际的值是由vm实例中的data属性提供的

```
<img v-bind:src='avatar' />
new Vue({
  data : {
    avatar : 'http://….'
  }
})
```

v-bind可以简写为：，上述例子即可简写为 `<img :src='avatar' />` 。

v-bind还拥有三种修饰符，分别为.sync、.once、.camel

- ~~.sync~~：用于组件props属性，进行双向绑定，即父组件绑定传递给子组件的值，无论在哪个组件中对其进行了修改，其他组件中的这个值也会随之更新。例如： `<my-child :parent.sync='parent'></my-child>` 。父组件实例vm.parent将通过prop选项传递给子组件my-child，即my-child组件构造函数需要定义选项props:['parent']，便可通过子组件自身实例vm.parent获取父组件传递的数据。两个组件都共享这一份数据，不论谁修改了这份数据，组件获取的数据都是一致的。但一般不推荐子组件直接修改父组件数据，这样会导致耦合且组件内的数据不容易维护。
- ~~.once~~：同.synce一样，用于组件props属性，但进行的是单次绑定。和双向绑定正好相反，单次绑定是将绑定数据传递给子组件后，子组件单独维护这份数据，和父组件的数据再无关系，父组件的数据发生变化也不会影响子组件中的数据。例如： `<my-child :parent.once='parent'></my-child>`
- .camel：将绑定的特性名字转回驼峰命名。只能用于普通HTML属性的绑定，通常会用于svg标签下的属性，例如： `<svg width='400' height='300' :view-box.camel='viewBox'></svg>` ，输出结果即为 `<svg width="400" height="300" viewBox="…..."></svg>`

不过在**Vue.js 2.0**中，修饰符**.syce**和**.once**均被废弃，规定组件间仅能单向传递，如果子组件需要修改父组件，则必须使用事件机制来进行处理。

3.1.2 v-model

该指令主要用于input、select、textarea标签中，具有lazy、number、debounce（2.0废除）、trim（2.0新增）这些修饰符。

3.1.3 v-if/v-else/v-show

v-if/v-else/v-show这三个指令主要用于根据条件展示对应的模板内容，这在第2.3.2小节的渲染语法中也进行了说明。

v-if和v-show的主要区别就在于，v-if在条件为false的情况下并不进行模板的编译，而v-show则会在模板编译好之后将元素隐藏掉。v-if的切换消耗要比v-show高，但初始条件为false的情况下，v-if的初始渲染要稍快

3.1.4 v-for

v-for也是用于模板渲染的指令，在第2.3.3小节列表渲染中我们也已说明过，这里就不再赘述。

v-for指令用法在Vue.js 2.0中做了些细微的调整，大致包含以下几个方面：

1. 参数顺序变化 当包含参数index或key时，对象参数修改为（item, index）或（value, key），这样与JS Array对象的新方法forEach和map，以及一些对象迭代器（例如lodash）的参数能保持一致。
2. v-bind:key 属性track-by被v-bind：key代替， `<div v-for="item in items" tranck-by="id">` 需要改写成 `<div v-for="item in items" v-bind:key="item.id">` 。
3. n in 10 v-for="n in 10"中的n由原来的0～9迭代变成1～10迭代。

3.1.5 v-on

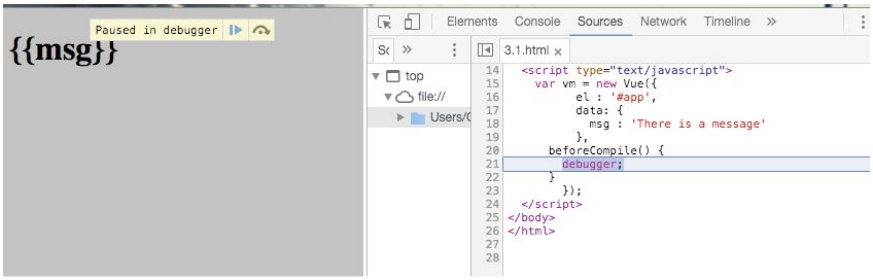
v-on指令主要用于事件绑定，在第2.4节中我们已经说明。 修饰符包括.stop、.prevent、.capture、.self以及指定按键.{keyCode|keyAlias}。

在**Vue.js 2.0**中，在组件上使用**v-on**指令只监听自定义事件，即使用**$emit**触发的事件；如果要监听原生事件，需要使用修饰符**.native**，例如 `<my-component v-on:click.native="onClick"></my-component>` 。

3.1.6 v-text

v-text，参数类型为String，作用是更新元素的textContent。{{}}文本插值本身也会被编译成textNode的一个v-text指令。而与直接使用{{}}不同的是，v-text需要绑定在某个元素上，能避免未编译前的闪现问题。例如：

`<span v-text="msg"></span>` 如果直接使用 `<span></span>`，在生命周期beforeCompile期间，此刻msg数据尚未编译至中，用户能看到一瞬间的{{msg}}，然后闪现为There is a message，而用v-text的话则不会有这个问题 如图所示：



3.1.7 v-HTML

v-HTML, 参数类型为String， 作用为更新元素的innerHTML，接受的字符串不会进行编译等操作，按普通HTML处理。同v-text类似，{{{}}}插值也会编译为节点的v-HTML指令，v-HTML也需要绑定在某个元素上且能避免编译前闪现问题。例如：

```
<div>\{\{\{HTML\}\}\}</div>
<div v-HTML="HTML"></div>
```

3.1.8 v-el

v-el指令为DOM元素注册了一个索引，使得我们可以直接访问DOM元素。语法上说，可以通过所属实例的$els属性调用。例如：

```
<div v-el:demo>there is a el demo</div>
vm.$els.demo.innerText // -> there is a el demo
```

或者在vm内部通过this进行调用。 另外，由于HMTL不区分大小写，在v-el中如果使用了驼峰式命名，系统会自动转成小写。但可以使用“-”来连接你期望大写的字母

```
<div v-el:camelCase>There is a camelcase</div>
<div v-el:camel-case>There is a camelCase</div>
vm.$els.camelcase.innerText // -> There is a camelcase
vm.$els.camelCase.innerText // -> There is a camelCase
```

3.1.9 v-ref

v-ref指令与v-el类似，只不过v-ref作用于子组件上，实例可以通过$refs访问子组件。命名方式也类似，想使用驼峰式命名的话用“-”来做连接。例如：

```
<message v-ref:title content="title"></message>
<message v-ref:sub-title content="subTitle"></message>
var Message = Vue.extend({
  props : ['content'],
  template : '<h1></h1>'
});
Vue.component('message', Message);
```

我们最终将vm.$refs.title和vm.$refs.subTitle用console.log的方式打印到控制台中，结果为：

输出了两个子组件的实例

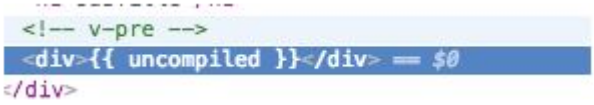
从理论上来说，我们可以通过父组件对子组件进行任意的操作，但实际上尽量还是会采用props数据绑定，用组件间通信的方式去进行逻辑上的交互，尽量让组件只操作自己内部的数据和状态，如果组件间有通信，也通过调用组件暴露出来的接口进行通信，而不是直接跨组件修改数据。

3.1.10 v-pre

v-pre指令相对简单，就是跳过编译这个元素和子元素，显示原始的{{}}Mustache标签，用来减少编译时间。例如：

```
<div v-pre></div>
var vm = new Vue({
  el : '#app',
  data: {
    uncompiled : 'Thers is an uncompiled element'
  }
});
```

最后输出：



3.1.11 v-cloak

v-cloak指令相当于在元素上添加了一个[v-cloak]的属性，直到关联的实例结束编译。官方推荐可以和css规则[v-cloak]{ display :none }一起使用，可以隐藏未编译的Mustache标签直到实例准备完毕。例如：

```
<div v-cloak></div>
```

3.1.12 v-once

v-once指令是Vue.js 2.0中新增的内置指令，用于标明元素或组件只渲染一次，即使随后发生绑定数据的变化或更新，该元素或组件及包含的子元素都不会再次被编译和渲染。这样就相当于我们明确标注了这些元素不需要被更新，所以v-once的作用是最大程度地提升了更新行为中页面的性能，可以略过一些明确不需要变化的步骤。使用方式如下：

```
<span v-once></span>
<my-component v-once :msg='msg'></my-component>
```

3.2 自定义指令基础

除了内置指令外，Vue.js也提供了方法让我们可以注册自定义指令，以便封装对DOM元素的重的处理行为，提高代码复用率。

3.2.1 指令的注册

我们可以通过Vue.directive(id, definition) 方法注册一个全局自定义指令，接收参数id和定义对象。id是指令的唯一标识，定义对象则是指令的相关属性及钩子函数。例如：

```
Vue.directive('global-directive', definition); // 我们暂时只注册了这个指令，并没有赋予这个
```

我们可以在模板中这么使用：

```
<div v-global-directive></div>
```

而除了全局注册指令外，我们也可以通过在组件的directives选项注册一个局部的自定义指令。例如：

```
var comp = Vue.extend({
  directives : {
    'localDirective' : {}  // 可以采用驼峰式命名
  }
});
```

该指令就只能在当前组件内通过v-local-directive的方式调用，而无法被其他组件调用。

3.2.2 指令的定义对象

我们在注册指令的同时，可以传入definition定义对象，对指令赋予一些特殊的功能。这个定义对象主要包含三个钩子函数：**bind**、 **update**和 **unbind**。

- bind: 只被调用一次，在指令第一次绑定到元素上时调用。
- update：指令在bind之后以初始值为参数进行第一次调用，之后每次当绑定值发生变化时调用，update接收到的参数为newValue和oldValue
- unbind：指令从元素上解绑时调用，只调用一次。

这三个函数都是可选函数，但注册一个空指令肯定没有意义，来看下面这个例子，会使我们对整个指令周期有更明确的认识。

```
<div v-if="isExist" v-my-directive="param"></div>
Vue.directive('my-directive', {
  bind : function() {
    console.log('bind', arguments);
  },
  update : function(newValue, oldValue) {
    console.log('update', newValue, oldValue)
  },
  unbind : function() {
    console.log('unbind', arguments);
  }
})
var vm = new Vue({
  el : '#app',
  data : {
    param : 'first',
    isExist : true
  }
});
```

我们在控制台里先后输入vm.param ='second'和vm.isExist = false，整体输出如下：

另外，如果我们只需要使用update函数时，可以直接传入一个函数代替定义对象：

```
Vue.directive('my-directive', function(value) {
  // 该函数即为update函数
});
```

上述例子中，可以使用my-directive指令绑定的值是data中的param属性。也可以直接绑定字符串常量，或使用字面修饰符，但这样的话需要注意update方法将只调用一次，因为普通字符串不能响应数据变化。

```
<div v-my-directive="constant string"/></div>  // -> value为undefined，因为data中没有对
<div v-my-direcitve="'constant string'"></div> // -> value 为constant string，绑定字符
<div v-my-directive.literal="constant string"></div> // -> value 为constant string，利
```

除了字符串外，指令也能接受对象字面量或任意合法的JavaScript表达式。例如：

```
<div v-my-directive="\{ title : 'Vue.js', author : 'You'\}"></div>
<div v-my-directive="isExist ? 'yes' : 'no'" ></div>
```

```
update Object {title: "Vue.js", author: "You"} undefined
update no undefined
```

注意此时对象字面量不需要用单引号括起来，这和字符串常量不一样。

3.2.3 指令实例属性

在指令的钩子函数内，可以通过this来调用指令实例

- el：指令绑定的元素。
- vm：该指令的上下文ViewModel，可以为new Vue()的实例，也可以为组件实例。
- expression：指令的表达式，不包括参数和过滤器。
- arg：指令的参数。
- name：指令的名字，不包括v-前缀。
- modifiers：一个对象，包含指令的修饰符。
- descriptor：一个对象，包含指令的解析结果。

我们可以通过以下这个例子，更直观地了解到这些属性：

```
<div v-my-msg:console.log="content"></div>
Vue.directive('my-msg', {
  bind : function() {
    console.log('~~~~~~~~~~~bind~~~~~~~~~~~~~~');
    console.log('el', this.el);
    console.log('name', this.name);
    console.log('vm', this.vm);
    console.log('expression', this.expression);
    console.log('arg', this.arg);
    console.log('modifiers', this.modifiers);
    console.log('descriptor', this.descriptor);
  },
  update : function(newValue, oldValue) {
    var keys = Object.keys(this.modifiers);
    window[this.arg][keys[0]](newValue);
  },
  unbind : function() {
  }
});
var vm = new Vue({
  el : '#app',
  data : {
    content : 'there is the content'
  }
});
```

输出结果如下：


bind~~~~~~~~~~~~~ 3.1.html:39
el 3.1.html:40
name my-msg 3.1.html:41
vm 3.1.html:42
▶ Vue {$el: div#app, $parent: undefined, $root: Vue, $children: Array[0], $refs: Object…}
expression content 3.1.html:43
arg console 3.1.html:44
modifiers Object {log: true} 3.1.html:45
descriptor 3.1.html:46
▶ Object {name: "my-msg", attr: "v-my-msg:console.log", raw: "content", def: Object, arg: "console"…}
there is the content 3.1.html:56

3.2.4 元素指令

元素指令是Vue.js的一种特殊指令，普通指令需要绑定在某个具体的DOM元素上，但元素指令可以单独存在，从使用方式上看更像是一个组件，但本身内部的实例属性和钩子函数是和指令一致的。例如：

```
<div v-my-directive></div> // -> 普通指令使用方式
<my-directive></my-directive> // -> 元素指令使用方式
```

元素指令的注册方式和普通指令类似，也有全局注册和局部注册两种。

```
Vue.elementDirective('my-ele-directive') // 全局注册方式
var Comp = Vue.extend({  // 局部注册，仅限该组件内使用
  … // 省略了其他参数
  elementDirectives : {
    'eleDirective' : {}
  }
});
Vue.component('comp', Comp);
```

元素指令不能接受参数或表达式，但可以读取元素特性从而决定行为。而且当编译过程中遇到一个元素指令时，Vue.js将忽略该元素及其子元素，只有元素指令本身才可以操作该元素及其子元素。
Vue.js 2.0中取消了这个特性，推荐使用组件来实现需要的业务。

3.3 指令的高级选项

Vue.js 指令定义对象中除了钩子函数外，还有一些其他的选项，我们将在本节中对其做逐个的讲述。

3.3.1 params

定义对象中可以接受一个params数组，Vue.js编译器将自动提取自定义指令绑定元素上的这些属性。例如：

```
<div v-my-advance-directive a="paramA"></div>
Vue.directive('my-advance-directive', {
  params : ['a'],
  bind : function() {
    console.log('params', this.params);
  }
});
```

```
params Object {a: "paramA"}
```

除了直接传入数值外，params支持绑定动态数据，并且可以设定一个watcher监听，当数据变化时，会调用这个回调函数。例如：

```
<div v-my-advance-directive v-bind:a="a"></div> // 当然也可以简写成
<div v-my-advance-directive :a="a"></div>
Vue.directive('my-advance-directive', {
  params : ['a'],
  paramWatchers : {
    a : function(val, oldVal) {
        console.log('watcher: ', val, oldVal)
    }
  },
  bind : function() {
    console.log('params', this.params);
  }
});
var vm = new Vue({
  el : '#app',
  data : {
    a : 'dynamitc data'
  }
});
```

```
params Object {a: "dynamic data"}
> vm.a = 123
watcher:  123 dynamic data
< 123
>
```

3.3.2 ~~deep~~

当自定义指令作用于一个对象上时，我们可以使用deep选项来监听对象内部发生的变化。例如：

```
<div v-my-deep-directive="obj"></div>
<div v-my-nodeep-directive="obj"></div>
Vue.directive('my-deep-directive', {
  deep : true,
  update : function(newValue, oldValue) {
    console.log('deep', newValue.a.b);
  }
});
Vue.directive('my-nodeep-directive', {
  update : function(newValue, oldValue) {
    console.log('deep', newValue.a.b);
  }
});
var vm = new Vue({
  el : '#app',
  data : {
    obj : {
      a : {
        b : 'inner'
      }
    }
  }
})
```

运行后，在控制台中输入vm.obj.a.b = 'inner changed', 只有my-deep-directive调用了update函数，输出了改变后的值。

```
  deep inner
  nodeep inner
> vm.obj.a.b = 'inner changed'
  deep inner changed
< "inner changed"
>
```

Vue.js 2.0中废弃了该选项。

3.3.3 twoWay

在自定义指令中，如果需要向Vue实例写回数据，就需要在定义对象中使用twoWay:true，这样可以在指令中使用this.set(value)来写回数据。

```
<input type="text" v-my-twoway-directive="param" / >
Vue.directive('my-twoway-directive', {
  twoWay : true,
  bind : function() {
    this.handler = function () {
      console.log('value changed: ', this.el.value);
      this.set(this.el.value)
    }.bind(this)
    this.el.addEventListener('input', this.handler)
  },
  unbind: function () {
    this.el.removeEventListener('input', this.handler)
  }
});
var vm = new Vue({
  el : '#app',
  data : {
    param : 'first',
  }
});
```

此时在input中输入文字，然后在控制台中输入vm.param即可观察到实例的param属性已被改变。

```
value changed:  1
value changed:  12
value changed:  123
value changed:  1231
value changed:  12312
value changed:  123123
vm.param
"123123"
```

需要注意的是，如果没有设定twoWay:true，就在自定义指令中调用this.set(), Vue.js会抛出异常。

```
▶ [Vue warn]: Directive.set() can only be used inside
twoWaydirectives.
>
```

3.3.4 acceptStatement

选项acceptStatement:true 可以允许自定义指令接受内联语句，同时update函数接收的值是一个函数，在调用该函数时，它将在所属实例作用域内运行。

```
<div v-my-directive="i++"></div>
Vue.directive('my-directive', {
  acceptStatement: true,
  update: function (fn) {
  }
})
var vm = new Vue({
  el : '#app',
  data : {
    i : 0
  }
});
```

如果在update函数中，运行fn()， 则会执行内联语句i++，此时vm.i = 1。但更改vm.i并不会触发update函数。 需要当心的是，如果此时没有设定acceptStatement: true，该指令会陷入一个死循环中。v-my-statement-directive接受到i的值每次都在变化，会重复调用update函数，最终导致Vue.js抛出异常。

3.3.5 terminal

选项terminal的作用是阻止Vue.js遍历这个元素及其内部元素，并由该指令本身去编译绑定元素及其内部元素。内置的指令v-if和v-for都是terminal指令。 使用terminal选项是一个相对较为复杂的过程，你需要对Vue.js的编译过程有一定的了解，这里借助官网的一个例子来大致说明如何使用terminal。

```
<div id="modal"></div>
...
<div v-inject:modal>
  <h1>header</h1>
  <p>body</p>
  <p>footer</p>
</div>
var FragmentFactory = Vue.FragmentFactory // Vue.js全局API，用来创造fragment的工厂函数，
var remove = Vue.util.remove // Vue.js工具类函数，移除DOM元素
var createAnchor = Vue.util.createAnchor  // 创建锚点，锚点在debug模式下是注释节点，非deb
Vue.directive('inject', {
  terminal: true,
  bind: function () {
    var container = document.getElementById(this.arg) // 获取需要注入到的DOM元素
    this.anchor = createAnchor('v-inject')  // 创建v-inject锚点
    container.appendChild(this.anchor) // 锚点挂载到注入节点中
    remove(this.el) // 移除指令绑定的元素
    var factory = new FragmentFactory(this.vm, this.el)  // 创建fragment
    this.frag = factory.create(this._host, this._scope, this._frag)
    // this._host 用于表示存在内容分发时的父组件
    // this._scope 用于表示存在v-for时的作用域
    // this._frag 用于表示该指令的父fragment
    this.frag.before(this.anchor)
  },
  unbind: function () {
    this.frag.remove()
    remove(this.anchor)
  }
})
```

最终我们得到的结果是：

```
<div id="modal">
    <div>
        <h1>header</h1>
        <p>body</p>
        <p>footer</p>
    </div>
</div>
```

3.3.6 priority

选项priority即为指定指令的优先级。普通指令默认是1000，termial指令默认为2000。同一元素上优先级高的指令会比其他指令处理得早一些，相同优先级则按出现顺序依次处理。以下为内置指令优先级顺序：

```
export const ON = 700
export const MODEL = 800
export const BIND = 850
export const TRANSITION = 1100
export const EL = 1500
export const COMPONENT = 1500
export const PARTIAL = 1750
export const IF = 2100
export const FOR = 2200
export const SLOT = 2300
```

3.4 指令在Vue.js 2.0中的变化

由于指令在Vue.js2.0中发生了比较大的变化，所以本节单独来说明下这些情况。总得来说，Vue.js 2.0中的指令功能更为单一，很多和组件重复的功能和作用都进行了删除，指令也更专注于本身作用域的操作，而尽量不去影响指令外的DOM元素及数据。

3.4.1 新的钩子函数

钩子函数增加了一个componentUpdated，当整个组件都完成了update状态后即所有DOM都更新后调用该钩子函数，无论指令接受的参数是否发生变化。

3.4.2 钩子函数实例和参数变化

在Vue.js 2.0中取消了指令实例这一概念，即在钩子函数中的this并不能指向指令的相关属性。指令的相关属性均通过参数的形式传递给钩子函数。

```
Vue.directive('my-directive', {
  bind : function(el, binding, vnode) {
    console.log('~~~~~~~~~~bind~~~~~~~~~~');
    console.log('el', el);
    console.log('binding', binding);
    console.log('vnode', vnode);
  },
  update : function(el, binding, vnode, oldVNode) {
    ….
  },
  componentUpdated(el, binding, vnode, oldVNode) {
    ….
  },
  unbind : function(el, binding, vnode) {
    ….
  }
});
```

```
~~~~~~~~~~bind~~~~~~~~~~
el ▶ div
binding ▶ Object {name: "my-directive", value: "first", expression: "param", modifiers: Object}
vnode ▶ VNode {tag: "div", data: Object, children: undefined, text: undefined, elm: div…}
```

在Vue.js 1.0的实例中的属性大部分都能在binding中找到，vnode则主要包含了节点的相关信息，有点类似于fragment的作用。

3.4.3 update函数触发变化

钩子函数update对比Vue.js 1.0也有了以下两点变化：① 指令绑定bind函数执行后不直接调用update函数。② 只要组件发生重绘，无论指令接受的值是否发生变化，均会调用update函数。如果需要过滤不必要的更新，则可以使用binding.value == binding.oldValue来判断。

3.4.4 参数binding对象

钩子函数接受的参数binding对象为不可更改，强行设定binding.value的值并不会引起实际的改动。如果非要通过这种方式进行修改的话，只能通过el直接修改DOM元素。

第 4 章 过滤器

Vue.js允许在表达式后面添加可选的过滤器，以管道符表示，例如：

``

过滤器的本质是一个函数，接受管道符前面的值作为初始值，同时也能接受额外的参数，返回值为经过处理后的输出值。多个过滤器也可以进行串联。例如：

```
\{\{ message | filterA 'arg1' 'arg2' \}\}
\{\{ message | filterA | filterB\}\}
```

4.1 过滤器注册

Vue.js提供了全局方法Vue.filter()注册一个自定义过滤器，接受过滤器ID和过滤器函数两个参数。例如：

```
Vue.filter('date', function(value) {
  if(!value instanceof Date) return value;
  return value.toLocaleDateString();
})
```

这样注册之后，我们就可以在vm实例的模板中使用这个过滤器了。

```
<div>
  2019-12-08T21:18:24+08:00
</div>
var vm = new Vue({
  el : '#app',
  data: {
    date : new Date()
  }
});
```

除了初始值之外，过滤器也能接受任意数量的参数。例如：

```
Vue.filter('date', function(value, format) {
  var o = {
    "M+": value.getMonth() + 1, //月份
    "d+": value.getDate(), //日
    "h+": value.getHours(), //小时
    "m+": value.getMinutes(), //分
    "s+": value.getSeconds(), //秒
  };

  if (/(y+)/.test(format))
    format = format.replace(RegExp.$1, (value.getFullYear() + "").substr(4 - RegExp.$1
  for (var k in o)
    if (new RegExp("(" + k + ")").test(format))
       format = format.replace(RegExp.$1, (RegExp.$1.length == 1) ? (o[k]) : (("00" + 
  return format;
});
```

使用方式即为：

```
<div>
  \{\{ date | date 'yyyy-MM-dd hh:mm:ss'\}\}  //->  2016-08-10 09:55:35 即可按格式输出当
</div>
```

4.2 双向过滤器

之前提及的过滤器都是在数据输出到视图之前，对数据进行转化显示，但不影响数据本身。Vue.js也提供了在改变视图中数据的值，写回data绑定属性中的过滤器，称为双向过滤器。例如：

```
<input type="text" v-model="price | cents" >
// 该过滤器的作用是处理价钱的转化，一般数据库中保存的单位都为分，避免浮点运算
Vue.filter('cents', {
  read : function(value) {
    return (value / 100).toFixed(2);
  },
  write : function(value) {
    return value * 100;
  }
});

var vm = new Vue({
  el : '#app',
  data: {
    price : 150
  }
});
```

从使用场景和功能来看，双向过滤器和第2章中提到的计算属性有点雷同。而Vue.js 2.0 中也取消了过滤器对v-model、v-on这些指令的支持，认为会导致更多复杂的情况，而且使用起来并不方便。所以**Vue.js 2.0中只允许开发者在{{}}标签中使用过滤器，像上述对写操作有转化要求的数据，建议使用计算属性这一特性来实现。**

4.3 动态参数

过滤器除了能接受单引号（"）括起来的参数外，也支持接受在vm实例中绑定的数据，称之为动态参数。使用区别就在于不需要用单引号将参数括起来。例如：

```
<input type="text" v-model="price" />
<span>\{\{ date | dynamic price \}\}</span>

Vue.filter('dynamic', function(date, price) {
  return date.toLocaleDateString() + ' : ' + price;
});
var vm = new Vue({
  el : '#app',
  data: {
    date : new Date(),
    price : 150
  }
});
```

过滤器中接受到的price参数即为vm.price。

4.4 过滤器在Vue.js 2.0中的变化

1. 取消了所有内置过滤器，即capitalize, uppercase, json等。 作者建议尽量使用单独的插件来按需加入你所需要的过滤器。不过如果你觉得仍然想使用这些Vue.js 1.0中的内置过滤器，也不是什么难办的事。1.0源码filters/目录下的index.js和array-filter.js中就是所有内置过滤器的源码，你可以挑选你想用的手动加到2.0中。
2. 取消了对v-model和v-on的支持，过滤器只能使用在{{}}标签中。
3. 修改了过滤器参数的使用方式，采用函数的形式而不是空格来标记参数。例如：{{ date | date('yyyy-MM-dd') }}。

第 5 章 过渡

过渡系统是Vue.js为DOM动画效果提供的一个特性，它能在元素从DOM中插入或移除时触发你的CSS过渡（transition）和动画（animation），也就是说在DOM元素发生变化时为其添加特定的class类名，从而产生过渡效果。除了CSS过渡外，Vue.js的过渡系统也支持javascript的过渡，通过暴露过渡系统的钩子函数，我们可以在DOM变化的特定时机对其进行属性的操作，产生动画效果。

5.1 CSS过渡

本小节首先来了解下CSS过渡的用法。

5.1.1 CSS过渡的用法

首先举一个例子来说明CSS过渡系统的使用方式：

```
<div v-if="show" transition="my-startup"></div>
var vm = new Vue({
  el : '#app',
  data: {
    show : false
  }
});
```

首先在模板中用transition绑定一个DOM元素，并且使用v-if指令使元素先处于未被编译状态。然后在控制台内手动调用vm.show = true, 就可以看到DOM元素最后输出为：

```
<div class="my-startup-transition"></div>
```

我们可以看到在DOM元素完成编译后，过渡系统自动给元素添加了一个my-startup-transition的class类名。那么为了让这个效果更明显一点，还可以提前给这个类名添加一点CSS样式：

```
.my-startup-transition {
  transition: all 1s ease;
  width: 100px;
  height: 100px;
  background: black;
  opacity: 1;
}
```

此时再重新刷新并手动运行vm.show = true，发现最终样式效果是加载上去了，但并没有出现transition的效果。这是由于在编译v-if后，div直接挂载到body并添加my-startup-transition类名这两个过程中浏览器仅进行了一次重绘，这对于div来说并没有产生属性的更新，所以没有执行css transition的效果。为了解决这个问题，Vue.js的过渡系统给元素插入及移除时分别添加了2个类名：*-enter*和-leave，*即为transition绑定的字符串，本例中即为my-startup。所以，在上述例子中，我们还需要添加两个类名样式，即my-startup-enter, my-startup-leave:

```
.my-startup-enter, .my-startup-leave{
  height: 0px;
  opacity: 0;
}
```

此时再重复之前的操作，就可以看到过渡效果了。需要注意的是，这两个类名的优先级需要高于.my-startup-transition，不然被my-startup-transition覆盖后就失效了。

同样，我们也可以通过CSS的animation属性来实现过渡的效果，例如：

```
<style>
  .my-animation-transition {
    animation: increase 1s ease 0s 1;
    width: 100px;
    height: 100px;
    background: black;
  }
  .my-animation-enter, .my-animation-leave {
    height: 0px;
  }
  @keyframes increase {
    from { height: 0px; }
    to { height: 100px; }
  }
</style>
```

```
<div v-if="animation" transition="my-animation">animation</div>
```

```
var vm = new Vue({
  el : '#app',
  data: {
    animation : false
  }
});
```

同样，更改vm.animation为true后即可看到过渡效果。 除了直接在元素上添加transition="name"外，Vue.js也支持动态绑定CSS名称，可用于元素需要多个过渡效果的场景。例如：

```
<div v-if="show" v-bind:transition=" transitionName"></div>
// 也可以简写成
<div v-if="show" :transition="transitionName"></div>
```

```
var vm = new Vue({
  el: '#app',
  data: {
    show: false,
    transitionName: 'fade'
  }
});
```

Vue.js本身并不提供内置的过渡CSS样式，仅仅是提供了过渡需要使用的样式的加载或移除时机，这样更便于我们灵活地按需去设计过渡样式。

5.1.2 CSS过渡钩子函数

Vue.js提供了在插入或DOM元素时类名变化的钩子函数，可以通过Vue.transition('name', {})的方式来执行具体的函数操作。例如：

```
Vue.transition('my-startup', {
  beforeEnter: function (el) {
    console.log('beforeEnter', el.className);
  },
  enter: function (el) {
    console.log('enter', el.className);
  },
  afterEnter: function (el) {
    console.log('afterEnter', el.className);
  },
  enterCancelled: function (el) {
    console.log('enterCancelled', el.className);
  },
  beforeLeave: function (el) {
    console.log('beforeLeave', el.className);
  },
  leave: function (el) {
    console.log('leave', el.className);
  },
  afterLeave: function (el) {
    console.log('afterLeave', el.className);
  },
  leaveCancelled: function (el) {
    console.log('leaveCancelled', el.className);
  }
})
```

在控制台里执行vm.show = true，输出结果如下：

```
vm.show = true
beforeEnter my-startup-transition
enter my-startup-transition my-startup-enter
true
afterEnter my-startup-transition
```

这样，我们能很清楚地看到钩子函数执行的顺序以及元素类名的变化。同样的，还可以再次更改vm.show的值置为false，结果如下：

```
vm.show = false
beforeLeave my-startup-transition
leave my-startup-transition my-startup-leave
false
afterLeave my-startup-transition
```

由于元素在使用CSS的transition和animation时，系统的流程不完全一样。所以先以transition为例，总结下过渡系统的流程。

当vm.show = true时，

1. 调用beforeEnter函数。

2. 添加enter类名到元素上。
3. 将元素插入DOM中。
4. 调用enter函数。
5. 强制reflow一次，然后移除enter类名，触发过渡效果。
6. 如果此时元素被删除，则触发enterCancelled函数。
7. 监听transitionend事件，过渡结束后调用afterEnter函数。

当vm.show = false 时，

1. 调用beforeLeave函数。
2. 添加v-leave类名，触发过渡效果。
3. 调用leave函数。
4. 如果此时元素被删除，则触发leaveCancelled函数。
5. 监听transitionend事件，删除元素及*-leave类名。
6. 调用afterLeave函数。

如果使用animation作为过渡的话，在DOM插入时，*-enter类名不会立即被删除，而是在animationend事件触发式删除。

另外，enter和leave函数都有第二个可选的回调参数，用于控制过渡何时结束，而不是监听transitionend和animationend事件，例如：

```
<style>
   my-done-transition {
    transition: all 2s ease;
    width: 100px; height: 100px;
    background: black;
    opacity: 1;
  }
  .my-done-enter, .my-done-leave{
    height: 0px;
    opacity: 0;
  }
</style>
```

```
Vue.transition('my-done', {
  enter: function (el, done) {
    this.enterTime = new Date();
    setTimeout(done, 500);
  },
  afterEnter: function (el) {
    console.log('afterEnter', new Date() - this.enterTime);
  }
})
var vm = new Vue({
  el : '#app',
  data: {
    done : false
  }
});
```

```
vm.done = true
true
afterEnter 500
```

此时afterEnter函数执行的时间就不是my-done-transition样式中的2s之后，而是done调用的500ms之后。需要注意的是，如果在enter和leave中声明了形参done，但没有调用，则不会触发afterEnter函数。

5.1.3 显示声明过渡类型

Vue.js可以指定过渡元素监听的结束事件的类型， 例如：

```
Vue.transition('done-type', {
  type: 'animation'
})
```

此时Vue.js就只监听元素的animationend事件，避免元素上还存在transition时导致的结束事件触发不一致。

5.1.4 自定义过渡类名

除了使用默认的类名*-enter,* -leave外，Vue.js也允许我们自定义过渡类名，例如：

```
Vue.transition('my-startup', {
  enterClass: 'fadeIn',
  leaveClass: 'fadeOut'
})
```

我们可以通过上述钩子函数的例子，观测元素的类名变化：

```
vm.show = true
beforeEnter my-startup-transition
enter my-startup-transition fadeIn
true
afterEnter my-startup-transition
vm.show = false
beforeLeave my-startup-transition
leave my-startup-transition fadeOut
false
afterLeave my-startup-transition
```

Vue.js官方推荐了一个CSS动画库，animate.css，配合自定义过渡类名使用，可以达到非常不错的效果。只需要引入一个CSS文件，http:\/\/cdn.bootcss.com\/animate.css\/3.5.2\/animate.min.css ，就可以使用里面的预设动画。例如：

```
<div v-if="animateShow" class="animated" transition="bounce">bounce effect</div>
```

```
Vue.transition('bounce', {
  enterClass: 'bounceIn',
  leaveClass: 'bounceOut'
})
```

在使用animate.css时，需要先给元素附上animated类名，然后再添加预设的动效类名，例如上例中的bounceIn、bounceOut，这样就能看到动画效果。这个库提供了多种强调展示（例如弹性、抖动）、渐入渐出、翻转、旋转、放大缩小等效果。所有的效果可以访问官方网址https:\/\/daneden.github.io\/animate.css\/ 在线观看。

5.2 JavaScript过渡

Vue.js也可以和一些JavaScript动画库配合使用，这里只需要调用JavaScript钩子函数，而不需要定义CSS样式。transition接受选项css:false，将直接跳过CSS检测，避免CSS规则干扰过渡，而且需要在enter和leave钩子函数中调用done函数，明确过渡结束时间。此处将引入Velocity.js来配合使用JavaScript过渡。

5.2.1 Velocity.js

Velocity.js是一款高效的动画引擎，可以单独使用也可以配合jQuery使用。它拥有和jQuery的animate一样的api接口，但比jQuery在动画处理方面更强大、更流畅，以及模拟了一些现实世界的运动，例如弹性动画等。 Velocity.js可以当做jQuery的插件使用，例如：

```
$element.velocity({ left: "100px"}, 500, "swing", function(){console.log("done")});
$element.velocity({ left: "100px"}, {
   duration: 500,
   easing: "swing",
   complete : function(){console.log("done");}
});
```

也可以单独使用，例如：

```
var el = document.getElementById(id);
Velocity(el, { left : '100px' }, 500, 'swing', done);
```

5.2.2 JavaScript过渡使用

我们可以通过以下方式注册一个自定义的JavaScript过渡：

```
.my-velocity-transition {
  position: absolute; top:0px;
  width: 100px; height: 100px;
  background: black;
}
</style>
```

```
<div v-if="velocity" transition="my-velocity"></div>
```

```
Vue.transition('my-velocity', {
  css : false,
  enter: function (el, done) {
    Velocity(el, { left : '100px' }, 500, 'swing', done);
  },
  enterCancelled: function (el) {
    Velocity(el, 'stop');
  },
  leave: function (el, done) {
    Velocity(el, { left : '0px' }, 500, 'swing', done);
  },
  leaveCancelled: function (el) {
    Velocity(el, 'stop');
  }
})
```

运行上述代码，在设置vm.velocity = true后，过渡系统即会调用enter钩子函数，通过Velocity对DOM操作展现动画效果，然后强制调用done函数，明确结束过渡效果。

5.3 过渡系统在Vue.js 2.0中的变化

过渡系统在Vue.js 2.0中也发生了比较大的变化，借鉴了ReactJS CSSTransitionGroup的一些相关设定和命名。

5.3.1 用法变化

新的过渡系统中取消了v-transition这个指令，新增了名为transition的内置标签，用法变更为：

```
<transition name="fade">
  <div class="content" v-if="show">content</div>
</transition>
```

transition标签为一个抽象组件，并不会额外渲染一个DOM元素，仅仅是用于包裹过渡元素及触发过渡行为。v-if、v-show等指令仍旧标记在内容元素上，并不会作用于transition标签上。

transition 标签能接受的参数与Vue.js 1.0中注册的transition接受的选项类似。

1. name
   同v-transition中接受的参数，自动生成对应的name-enter, name-enter-active类名。
2. appear
   元素首次渲染的时候是否启用transition，默认值为false。即v-if绑定值初始为true时，首次渲染时是否调用transition效果。在Vue.js 1.0中，v-if如果初始值为true的话，首次渲染是无法使用transition效果的，只有v-show能使用。
3. css
   同Vue.js 1.0的CSS选项，如果设置为true，则只监听钩子函数的调用。
4. type
   同Vue.js 1.0的type选项，设置监听CSS动画结束事件的类型。
5. mode
   控制过渡插入/移除的先后顺序，主要用于元素切换时。可供选择的值有“out-in”, “in-out”，如果不设置，则同时调用。例如：

   ```
   <transition name="fade" mode="out-in ">
     <p :key="ok"></p>  // 这里的:key='ok'主要用于强制替换元素，展现出in-out/out-in效果
   </transition>
   ```

   当ok在true和false切换时，mode=“out-in”决定先移除\false\<\/p>，等过渡结束后，再插入\true\<\/p>元素，mode=“in-out”则相反。

6. 钩子函数
   enterClass, leaveClass, enterActiveClass, leaveActiveClass, appearClass, appearActiveClass，可以分别自定义各阶段的class类名。

总得来说，在Vue.js 2.0中我们可以直接使用transition标签并设定其属性来定义一个过渡效果，而不需要像在Vue.js 1.0中通过Vue.transition()语句来定义。例如：

```
<transition
   name="fade"
   mode="out-in"
   appear
   @before-enter="beforeEnter"
   @enter="enter"
   @after-enter="afterEnter"
   @appear="appear"

   @before-leave="beforeLeave"
   @leave="leave"
   @after-leave="afterLeave"
   @leave-cancelled="leaveCancelled"
   >
     <div class="content" v-if="ok"></div>
</transition>
```

5.3.2 类名变化

从上述属性的变化中我们可以看出，Vue.js 2.0 中新增了两个类名 enter-active和leave-active，用于分离元素本身样式和过渡样式。我们可以把过渡样式放到*-enter-active*、-leave-active中，*-enter*，-leave中则定义元素过渡前的样式，而元素原本的样式则由自己的类名去控制，不和过渡系统自动添加的类名样式混合起来。举个例子：

```
content {
  width: 100px; height: 100px;
  background: black; opacity: 1;
}
.fade-enter, .fade-leave-active{
   opacity: 0;
}
.fade-enter-active, .fade-leave-active{
   transition: all 3s ease;
}
```

```
<transition name="fade">
  <div class="content" v-if="ok"></div>
</transition>
```

这样，`<transition fade="name"></transition>` 就可以当做一个可复用的过渡元素，作用到你期望的元素上。

enter-active添加到元素上的时机是在元素插入DOM树后，在transition/animation结束后从元素上移除。leave-active则在DOM元素开始移除时添加上，在transition/animation结束后移除。

5.3.3 钩子函数变化

Vue.js 2.0中添加了三个新的钩子函数，before-appear, appear和after-appear。appear主要是用于元素的首次渲染，如果同时声明了enter和appear的相关钩子函数，元素首次渲染的时候会使用appear系钩子函数，再次渲染的时候才使用enter系钩子函数。例如：

```
<transition
   name="fade" mode="in-out" appear
   @before-enter="beforeEnter"
   @enter="enter"
   @after-enter="afterEnter"

   @appear="appear"

   @before-leave="beforeLeave"
   @leave="leave"
   @after-leave="afterLeave"
>
   <div class="content" v-if="ok"></div>
</transition>
```

```
var vm = new Vue({
   el : '#app',
   data : {
       ok : true
   },
   methods : {
     beforeEnter : function(el) {
       console.log('beforeEnter', el.className);
     },
     enter : function(el) {
       console.log('enter', el.className);
     },
     afterEnter : function(el) {
       console.log('afterEnter', el.className);
     },
     appear : function(el) {
       console.log('appear', el.className);
     },
    beforeLeave : function(el) {
       console.log('beforeLeave', el.className);
     },
     leave : function(el) {
      console.log('leave', el.className);
     },
     afterLeave : function(el) {
       console.log('afterLeave', el.className);
     },
   }
});
```

```
beforeEnter content
appear content fade-enter fade-enter-active
afterEnter content
vm.ok = false
beforeLeave content
leave content fade-leave fade-leave-active
false
afterLeave content
vm.ok = true
beforeEnter content
enter content fade-enter fade-enter-active
true
afterEnter content
```

另外，取消了v-if时的leave-cancelled，元素一旦被移除则不能停止该操作。但使用v-show时，leave-cancelled钩子仍然有效。我们可以沿用上面这个例子，在transition上加一个钩子函数

\@leave-cancelled="leaveCancelled"，将元素设置成\

{{ ok }}\<\/div>，\
{{ ok }}\<\/div>两种情况，然后手动设置vm.ok = false, 并在元素仍在过渡中设置vm.ok = true。在v-if情况下，元素继续进行leave transition，在transition结束后再次进行enter过渡；而在v-show情况下，元素则直接停止了leave transition，并调用了leaveCancelled钩子函数，然后直接进行了enter过渡。

5.3.4 transition-group

除了内置的transition标签外，Vue.js 2.0提供了transition-group标签，方便作用到多个DOM元素上。例如：

```
<transition-group tag="ul" name="list">
   <li v-for="item in items" :key="item.id">

   </li>
</transition-group>
```

transition-group的主要作用是给其子元素设定一个统一的过渡样式，而不需要给每单个元素都用transition包裹起来。

和transition标签不一样，transition-group不是一个虚拟DOM，会真实渲染在DOM树中。默认会是span标签，但我们也可以通过属性tag来设定，如上例中transition-group最终输出的会是一个ul标签。另外，我们也可以通过\

这样的写法来设定标签。

transition-group接收的参数和transition基本一致，但不支持mode参数，而且每个transition-group的子元素都需要包含唯一的key，如上例中的key=item.id。

我们可以补全上面的代码，作为一个完整的transition-group例子：

```
<style>
   .list-li {
      width: 100px; height: 20px;
      transform: translate(0, 0);
   }
    .list-enter, .list-leave-active{
      opacity: 0; transform: translate(-30px, 0);
   }
   .list-enter-active, .list-leave-active{
      transition: all 0.5s ease;
   }
</style>
```

```
<transition-group tag="ul" name="list" appear>
    <li v-for="item in items" :key="item.id" class="list-li">

    </li>
</transition-group>
```

```
var vm = new Vue({
  el : '#app',
  data : {
      items : [
        { id : 1, text : '11' },
        { id : 2, text : '22' },
        { id : 3, text : '33' },
        { id : 4, text : '44' }
      ]
  }
})
```

我们可以在控制台里对vm.items进行push或splice操作，这样就能看到li标签的过渡效果了。

第 6 章 组件

代码复用一直是软件开发中长期存在的一个问题，每个开发者都想再次使用之前写好的代码，又担心引入这段代码后对现有的程序产生影响。从jQuery开始，我们就开始通过插件的形式复用代码，到Requirejs开始将js文件模块化，按需加载。这两种方式都提供了比较方便的复用方式，但往往还需要自己手动加入所需的CSS文件和HTML模块。现在，Web Components的出现提供了一种新的思路，可以自定义tag标签，并拥有自身的模板、样式和交互。Angularjs的指令，Reactjs的组件化都在往这方面做尝试。同样，Vue.js也提供了自己的组件系统，支持自定义tag和原生HTML元素的扩展。

6.1 组件注册

Vue.js创建组件构造器的方式非常简单，在本书第2.5章的时候也提到过：

```
var MyComponent = Vue.extend({ … });
```

这样，我们就获得了一个组件构造器，但现在还无法直接使用这个组件，需要将组件注册到应用中。Vue.js提供了两种注册方式，分别是全局注册和局部注册。

6.1.1 全局注册

全局注册需要确保在根实例初始化之前注册，这样才能使组件在任意实例中被使用，注册方式如下：

```
Vue.component('my-component', MyComponent);
```

这条语句需要写在var vm = new Vue({...})之前，注册成功之后，就可以在模块中以自定义元素\的形式使用组件。对于组件的命名，W3C规范是字母小写且包含一个短横杠“-”，Vue.js暂不强制要求，但官方建议遵循这个规则比较好。 整个使用方法代码如下：

```
<div id="app">
  <my-component></my-component>
</div>
```

```
var MyComponent = Vue.extend({
  template : '<p>This is a component</p>'
})

Vue.component('my-component', MyComponent)

var vm = new Vue({
  el : '#app'
});
```

输出结果如下：

```
<div id="app">
  <p>This is a component</p>
</div>
```

6.1.2 局部注册

局部注册则限定了组件只能在被注册的组件中使用，而无法在其他组件中使用，注册方式如下：

```
var Child = Vue.extend({
  template : '<p>This is a child component</p>'
});

var Parent = Vue.extend({
  template: '<div> \
    <p>This is a parent component</p> \
    <my-child></my-child> \
    </div>',
  components: {
    'my-child': Child
  }
});
```

输出结果即为：

```
<div>
  <p>This is a parent component</p>
  <p>This is a child component</p>
</div>
```

而如果在根实例中调用</my-child>，则会抛出异常 [Vue warn]: Unknown custom element: - did you register the component correctly? For recursive components, make sure to provide the "name" option.

6.1.3 注册语法糖

Vue.js对于上述两种注册方式也提供了简化的方法，我们可以直接在注册的时候定义组件构造器选项，例如：

```
// 全局注册
Vue.component('my-component', {
  template : '<p>This is a component</p>'
})
// 局部注册
var Parent = Vue.extend({
  template: '<div> \
      <p>This is a parent component</p> \
      <my-child></my-child> \
    </div>',
  components: {
    'my-child': {
       template : '<p>This is a child component</p>'
    }
  }
});
```

6.2 组件选项

组件接受的选项大部分与Vue实例一样，相同的部分本章就不赘述了，主要说明一下两者的区别和组件选项中的props，用于接受父组件传递的参数。

6.2.1 组件选项中与Vue选项的区别

组件选项中的el和data与Vue构造器选项中这两个属性的赋值会稍微有些不同。在Vue构造器中是直接赋值：

```
var vm = new Vue({
  el : '#app',
  data : {
    name : 'Vue'
  }
})
```

而在组件中需要这么定义：

```
var MyComponent = Vue.extend({
  data : function() {
    return {
        name : 'component'
    }
  }
})
```

这是因为MyComponent可能会拥有多个实例，例如在某个组件模板中多次使用</my-component>。如果将对象data直接传递给了Vue.extend({})，那所有MyComponent的实例会共享一个data对象，所以需要通过函数来返回一个新对象。同样，el也是这么处理，只不过在组件中通过el来直接设定挂载元素的情况比较少见，自然避免了这种情况。

也就是说组件中data的赋值是通过函数返回

6.2.2 组件Props

选项props是组件中非常重要的一个选项，起到了父子组件间桥梁的作用。

首先，需要明确的是，组件实例的作用域是孤立的，也就是说子组件的模板和模块中是无法直接调用父组件的数据，所以通过props将父组件的数据传递给子组件，子组件在接受数据时需要显式声明props，例如：

```
Vue.component('my-child', {
  props : ['parent'],
  template: '<p> is from parent'
})
<my-child parent="This data"></my-child>  //-> <p>This data is from parent </p>
```

这就是props的基本使用方法，另外还有几点细节会进行详细说明。

1. 驼峰命名
   同指令等情况相同，由于HTML属性不区分大小写，如果我们在中的属性使用驼峰式myParam命名，即，那在props中的命名即为props: ['myparam']。所以如果需要使用驼峰式命名的话，我们需要在标签中使用my-param，用“-”的方式隔开，这样在props中就可以使用props: ['myParam']的形式进行声明。

2. 动态Props
   除了上述例子中传递静态数据的方式外，我们也可以通过v-bind的方式将父组件的data数据传递给子组件，例如：

   ```
   <div id="app">
     <input type="text" v-model="message" />
     <my-component v-bind:message="message"></my-component>
   </div>
   var MyComponent = Vue.extend({
     props : ['message'],
     template : "<p>From parent : undefined</p>"
   })

   Vue.component('my-component', MyComponent);

   var vm = new Vue({
     el : '#app',
     data : {
       message : 'default'
     }
   });
   ```

   这样我们在更改根实例message的值的时候，组件中的值也随之改动。除了v-bind外，也可以直接简写成 `<my-component :message="message"></my-component>` 。

   需要注意的是如果直接传递一个数值给子组件，就必须借助动态Props。如果通过\\<\/my-component>这种方式传递的话，则在子组件中获取的message其实是字符串“1”，只有通过如下的方式，才能准确传递数值：

```
<my-num :num="num"></my-num>
Vue.component('my-num', {
  props : ['num'],
  template : "<p>\{\{ num + ' is a ' + typeof num \}\}</p>",
});
var vm = new Vue({
  el : '#app',
  data : {
    num : 1
  }
});
// 输出 <p>1 is a number</p>
```

3. 绑定类型
   在动态绑定中，v-bind指令也提供了几种修饰符来进行不同方式的绑定。Props绑定默认是单向绑定，即当父组件的数据发生变化时，子组件的数据随之变化，但在子组件中修改数据并不影响父组件。修饰符.sync和.once显示的声明绑定为双向或单次绑定，例如：

```
<div id="app">
  <div>
      Parent component: <input type="text" v-model="msg" />
  </div>
  <my-bind :msg="msg"></my-bind>
</div>

Vue.component('my-bind', {
    props : ['msg'],
    template : '<div> \
        Child component : \
        <input type="text" v-model="msg"/> \
    </div>'
  });

var vm = new Vue({
  el : '#app',
  data : {
    msg : ''
  }
});
```

   此时父子组件中的即是单向绑定，可以通过input修改子组件中的值并不影响父组件中的值。

   而如果将上述例子中 `<my-bind :msg="msg"></my-bind>` 替换成 `<my-bind :msg.sync="msg"></my-bind>`，则在子组件的input中修改值即会影响父组件的值。

   once修饰符意味着单次绑定，子组件接受一次父组件传递的数据后，单独维护这份数据，既不影响父组件数据也不受其影响而更新。

   需要注意的是，由于Vue.js处理的方式是引用传递，所以如果prop传递的是一个对象或数组，那在子组件内进行修改就会影响父组件的状态，即使是单向绑定也一样。

4. Props验证
   组件可以指定props验证要求，这对开发第三方组件来说，可以让使用者更加准确地使用组件。使用验证的时候，props接受的参数为json对象，而不是上

述例子中的数组，例如：props : { a : Number }，即为验证参数a需为Number类型，如果调用该组件传入的a参数为字符串，则会抛出异常。Vue.js提供的Props验证方式有很多种，下面逐一进行说明：

1）基础类型检测：prop: Number，接受的参数为原生构造器，String、Number、Boolean、Function、Object、Array。也可接受null，意味任意类型均可。

2）多种类型：prop:[Number, String]， 允许参数为多种类型之一，例如类型可以为数值或字符串。

3）参数必需：prop: { type : Number, required: true}，参数必须有值且为Number类型。

4）参数默认：prop: { type : Number, default : 10 }，参数具有默认值10。需要注意的是，如果默认值设置为数组或对象，需要像组件中data属性那样，通过函数返回值的形式赋值，如：

```
prop : {
  type : Object,
  default : function() {
    return { a : 'a' }
  }
}
```

5）绑定类型：prop: { twoWay : true}，校验绑定类型，如果非双向绑定会抛出一条警告。

6）自定义验证函数：prop : { validator : function(value) { return value > 0; } }，验证值必须大于0。

7）转换值：prop: { coerce : function(val) { return parseInt(val) }}，将字符串转化成数值。

在开发环境中，如果验证失败了，Vue将抛出一条警告，组件上也无法设置此值。

在Vue.js 2.0中，验证属性twoWay和coerce均被废弃。
由于组件间只支持单向绑定，所有twoWay的校验也就不存在了。而coerce的用法和计算属性过于类似，所以也被废弃，官方推荐使用计算属性代替。

6.3 组件间通信

组件间通信是组件开发时非常重要的一环，我们既希望组件的独立性，数据能互不干涉，又不可避免组件间会有联系和交互。Vue.js在组件间通信这一部分既提供了直接访问组件实例的方法，也提供了自定义事件机制，通过广播、派发、监听等形式进行跨组件的函数调用。

6.3.1 直接访问

在组件实例中，Vue.js提供了以下三个属性对其父子组件及根实例进行直接访问。

1. $parent: 父组件实例。
2. $children: 包含所有子组件实例。
3. $root: 组件所在的根实例。

这三个属性都挂载在组件的this上，虽然Vue.js提供了直接访问这种方式，但我们并不提倡这么操作。这会导致父组件和子组件紧密耦合，且自身状态难以理解，所以建议尽量使用props在组件间传递数据。

6.3.2 自定义事件监听

在Vue实例中，系统提供了一套自定义事件接口，用于组件间通信，方便修改组件状态。类似于在jQuery，我们给DOM元素绑定一个非原生的事件，例如：$('#ele').on('custom', fn)，然后通过手动调用$('#ele').trigger('custom')方式来进行事件的触发。

那在Vue.js中，我们先来看下如何在实例中监听自定义事件。

1. events选项
   我们可以在初始化实例或注册子组件的时候，直接传给选项events一个对象，例如：

```
var vm = new Vue({
  el : '#app',
  data : {
    todo : []
  },
  events : {
    'add' : function(msg) {
       this.todo.push(msg);
    }
  }
});
```

2. $on方法
   我们也可以在某些特定情况或方法内采用$on方法来监听事件，例如：

```
var vm = new Vue({
  el : '#app',
  data : {
    todo : []
  },
  methods : {
    begin : function() {
      this.$on('add', function(msg) {
         this.todo.push(msg);
      });
    }
  }
});
```

6.3.3 自定义事件触发机制

设置完成事件监听后，下面来看下Vue.js的触发机制。

1. $emit
   在实例本身上触发事件。例如：

```
events : {
  'add' : function(msg) {
    this.todo.push(msg);
  }
}
methods: {
  onClick : function() {
    this.$emit('add', 'there is a message');// 即可触发events中的add函数
  }
}
```

2. $dispatch
   派发事件，事件沿着父链冒泡，并且在第一次触发回调之后自动停止冒泡，除非触发函数明确返回true，才会继续向上冒泡。

   父组件：

```
events : {
  'add' : function(msg) {
    this.todo.push(msg);
    // return true  明确返回true后，事件会继续向上冒泡
  }
}
```

   子组件：

```
methods: {
  toParent : function() {
    this.$dispatch('add', 'message from child');
  }
}
```

   调用子组件中的toParent()函数，即可向上冒泡，触发父组件中定义好的add事件。

3. $broadcast
   广播事件，事件会向下传递给所有的后代。例如：

   父组件：

```
methods: {
  toChild : function() {
    this.$dispatch('msg', 'message from parent');
  }
}
```

   子组件：

```
  events : {
    'msg' : function(msg) {
      alert(msg);
    }
  }
```

下面，我们可以通过这个完整的例子来验证这三种触发机制：

```
<div id="app">
  <input type="text" v-model="content">
  <button @click="addTodo">添加</button>
  <button @click="broadcast">广播</button>
  <child-todo name="one"></child-todo>
  <child-todo name="two"></child-todo>
  <ul>
    <li v-for="value in todo">

    </li>
  </ul>
</div>
```

// 子组件

```
Vue.component('child-todo', {
  props : ['name'],
  data : function() {
    return {
        content : ''
    }
  },
  template : '<div>\
        Child  \
        <input type="text" v-model="content"/> \
        <button @click="add">添加</button> \
    </div>',
  methods : {

    add : function() {
        // 将事件向上派发，这样既修改了父组件中的todo属性，又不直接访问父组件
        this.$dispatch('add', 'Child ' + this.name + ': ' + this.content);
        this.content = '';
    }
  },
  events : {
    // 用于接收父组件的广播
    'to-child' : function(msg) {
        this.$dispatch('add', 'Child ' + this.name + ': ' + msg);
    }
  }
});
```

// 根实例

```
var vm = new Vue({
  el : '#app',
  data : {
    todo : [],
    content : ''
  },
  methods : {
    addTodo : function() {
        // 触发自己实例中的事件
        this.$emit('add', 'Parent: ' + this.content);
        this.content = '';
    },
    broadcast : function() {
        // 将事件广播，使两个子组件实例都触发to-child事件
        this.$broadcast('to-child', this.content);
        this.content = '';
    }
  },
  events : {
    'add' : function(msg) {
        this.todo.push(msg);
    }
  }
});
```

在根实例的input中输入内容“Hello”，点击“添加”， 即会在绑定v-for指令的li标签中输出“Parent：Hello”；或点击“广播”，则会输出“Child one: hello”和“Child two: hello”两条数据，本质就是广播了事件to-child，两个子组件接受后触发了监听函数，将内容和组件name参数添加到父组件的todo数组中。

6.3.4 子组件索引

虽然我们不建议组件间直接访问各自的实例，但有时也不可避免，Vue.js也提供了直接访问子组件的方式。除了之前的this.children外，还可以给子组件绑定一个v-ref指令，指定一个索引ID，例如：

```
<child-todo v-ref:first></child-todo>
```

这样，在父组件中就可以通过this.$refs.first的方式获取子组件实例。

另外，如果v-ref作用在v-for绑定的元素上，例如：`<li v-for="item" v-ref:items></li>`，父组件获取的this.$refs.items为一个数组，包含相应的子组件实例。

6.4 内容分发

在实际的一些情况中，子组件往往并不知道需要展示的内容，而只提供基础的交互功能，内容及事件由父组件来提供。例如我们经常会使用的Bootstrap的模态框（Modal）插件

我们在调用时只希望使用Modal的浮层属性，以及显示/关闭浮层等控制函数，但内容本身则由父组件来决定。对此Vue.js提供了一种混合父组件内容与子组件自己模板的方式，这种方式称之为内容分发。Vue.js参照了当前web component规范草稿，使用元素为原始内容的插槽。首先解释下内容分发的一些基础用法和概念，最后来说明下Modal这个实际案例。

6.4.1 基础用法

先举一个简单的例子来说明内容分发的基础用法：

```
<div id="app">
  // 使用包含slot标签属性的子组件
  <my-slot>
    // 属性slot值需要与子组件中slot的name值匹配
    <p slot="title"></p>
    <div slot="content"></div>
  </my-slot>
</div>
// 注册my-slot组件，包含<slot> 标签，且设定唯一标识name
Vue.component('my-slot', {
  template : '<div>\
    <div class="title"> \
      <slot name="title"></slot> \
    </div> \
    <div class="content"> \
      <slot name="content"></slot> \
    </div> \
  </div>',
});
var vm = new Vue({
  el : '#app',
  data : {
    title : 'This is a title',
    content : 'This is the content'
  }
});
```

// 最后输出结果

```
<div>
  <div class="title">
    <p slot="title">This is a title</p>
  </div>
  <div class="content">
    <div slot="content">This is the content</div>
  </div>
</div>
```

从上述例子中可以看出，父组件中的内容代替了子组件中的slot标签，使得我们可以在不同地方使用子组件的结构而且填充不同的父组件内容，从而提升组件的复用性。

6.4.2 编译作用域

在上述例子中，我们在父组件中调用\组件，并在\

{{ title }}\<\/p>中绑定数据title，从结果得知此时绑定的是父组件的数据。也就是说在这种\{{ data }}\<\/my-slot>模板情况下，父组件模板的内容在父组件作用域内编译；子组件模板的内容在子组件作用域内编译。

以下这样的父组件模板例子就是无效的：

```
<my-scope-slot>
  <p slot="title"></p>
</my-scope-slot>
Vue.component('my-scope-slot', {
  template : '<div>\
      <p>child: undefined</p> \
      <slot name="title"></slot> \
    </div>',
  data() {
    return {
      childData : 'child scope'
    }
  }
});
```

输出结果：

```
<div>
  <p>child: child scope</p>
  <p slot="title"></p>
</div>
```

6.4.3 默认slot

\标签允许有一个匿名slot，不需要有name值，作为找不到匹配的内容片段的回退插槽，如果没有默认的slot，这些找不到匹配的内容片段将被忽略。下面修改一下上面的例子：

```
<anonymous-slot>
  // 去除slot属性
  <div id="content"></div>
  <p slot="title"></p>
</anonymous-slot>
// 匿名slot
Vue.component('anonymous-slot', {
  template : '<div>\
    <div class="title"> \
       <slot name="title"></slot> \
    </div> \
    <div class="content"> \
       <slot></slot> \
    </div> \
  </div>',
});
```

此时id为content的元素即为找不到匹配的内容片段，由于我们在anonymous-slot组件中设置了匿名slot，所以Vue.js会把该元素插入到slot中，最后输出结果：

```
<div>
  <div class="title">
    <p slot="title">This is a title</p>
  </div>
  <div class="content">
    <div>This is the content</div>
  </div>
</div>
```

如果将子组件中的匿名\\<\/slot>替换成\\<\/slot>，则#content元素就直接被忽略了，输出结果为：

```
<div>
  <div class="title">
    <p slot="title">This is a title</p>
  </div>
  <div class="content">
  </div>
</div>
```

6.4.4 slot属性相同

在父组件中，我们也可以定义多个相同slot属性的DOM标签，这样会依次插入到对应的子组件的slot标签中，以兄弟节点的方式呈现，我们可以将上例中父组件的实例模板改成：

```
<my-slot>
  <p slot="title">undefined1</p>
  <p slot="title">undefined2</p>
  <div slot="content"></div>
</my-slot>
// 输出结果
<div>
  <div class="title">
    <p slot="title">This is a title1</p>
    <p slot="title">This is a title2</p>
  </div>
  <div class="content">
    <div slot="content">This is the content</div>
  </div>
</div>
```

6.4.5 Modal实例

本小节我们会通过Modal案例来演示内容分发的实际使用场景。 首先注册Modal子组件：

// Modal 组件模板

```
<script id="modalTpl" type="x-template">
<div role="dialog">
  <div role="document" v-bind:style="{width: optionalWidth}">
    <div class="modal-content">
      <slot name="modal-header">
        <div class="modal-header">
          <button type="button" class="close" @click="close"> <span>&times;</span></b
          <h4 class="modal-title" >
            <slot name="title">
              {{title}}
            </slot>
          </h4>
        </div>
      </slot>
      <slot name="modal-body">
        <div class="modal-body"></div>
      </slot>
      <slot name="modal-footer">
        <div class="modal-footer">
          <button type="button" class="btn btn-default" @click="close">取消</button>
          <button type="button" class="btn btn-primary" @click="callback">确定</butto
        </div>
      </slot>
    </div>
  </div>
</div>
</script>
```

// 注册Modal组件

```
Vue.component('modal', {
    template : '#modalTpl',  // 获取模板中的HTML结构
    props : {
      title: {  // Modal 标题
        type: String,
        default: ''
      },
      show: { // 控制Modal是否显示
        required: true,
        type: Boolean,
        twoWay: true
      },
      width: {  // Modal 宽度
        default: null
      },
      callback: {  // 点击确定按钮的回调函数
        type: Function,
        default : function() {}
      }
    },
    computed: {  // 计算属性
      optionalWidth () {  // 处理props的width属性
        if (this.width === null) {
          return null;
        } else if (Number.isInteger(this.width)) {
          return this.width + 'px';
        }
        return this.width;
      }
    },
    watch: {
      show (val) { // show值变化时调用该函数
        var el = this.$el;
        if (val) {
          el.style.display = 'block';  //show值为true时，显示根元素
        } else {
          el.style.display = 'none';  //show值为false时，隐藏根元素
        }
      }
    },
    methods: {
      close () {
        this.show = false;
      }
    }
  })
```

// 父组件调用方式

// 需要引入 http://cdn.bootcss.com/bootstrap/3.3.6/css/bootstrap.css样式

// 父组件中使用modal组件

```
<div id="app">
    <button @click="show = true">open</button>
    <modal :show.sync="show" width="300px" :callback="close">
      <!—替换modal组件中的<slot name="modal-header"></slot>插槽 -->
      <div slot="modal-header" class="modal-header">Title</div>
      <!—替换modal组件中的<slot name="modal-body"></slot>插槽 -->
      <div slot="modal-body" class="modal-body">
        <div class="inner">
          Content
        </div>
      </div>
      <!—由于父组件中没有设定slot="modal-footer"的元素，所以使用子组件中的默认HTML结构 -->
    </modal>
  </div>
```

```
var vm = new Vue({
    el : '#app',
    data : {
      show : false
    },
    methods : {
      close : function() {
        alert('save');
        this.show = false;
      }
    }
  });
```

最终得到一个被button控制打开的Modal模态框，并且内容由父组件定义，并提供模态框的宽度及确定后的回调函数。

6.5 动态组件

Vue.js支持动态组件，即多个组件可以使用同一挂载点，根据条件来切换不同的组件。使用保留标签\，通过绑定到is属性的值来判断挂载哪个组件。这种场景往往运用在路由控制或者tab切换中。本小节先介绍下动态组件的基础用法。

6.5.1 基础用法

我们通过一个切换页面的例子来说明一下动态组件的基础用法：

```
<div id="app">
  // 相当于一级导航栏，点击可切换页面
  <ul>
    <li @click="currentView = 'home'">Home</li>
    <li @click="currentView = 'list'">List</li>
    <li @click="currentView = 'detail'">Detail</li>
  </ul>
  <component :is="currentView"></component>
</div>
var vm = new Vue({
  el : '#app',
  data: {
    currentView: 'home'
  },
  components: {
    home: {
      template : '<div>Home</div>'
    },
    list: {
        template : '<div>List</div>'
    },
    detail: {
        template : '<div>Detail</div>'
    }
  }
});
```

component标签上is属性决定了当前采用的子组件，:is是v-bind:is的缩写，绑定了父组件中data的currentView属性。顶部的ul则起到导航的作用，点击即可修改currentView值，也就修改component标签中使用的子组件类型，需要注意的事，currentView的值需要和父组件实例中的components属性的key相对应。

6.5.2 keep-alive

component标签接受keep-alive属性，可以将切换出去的组件保留在内存中，避免重新渲染。我们将上述例子中的component标签修改为：

```
<component :is="currentView" keep-alive></component>
```

并且将home组件修改为：

```
home: {
  template : '<div> \
      <p>Home</p> \
        <ul> \
          <li v-for="item in items">{{ item }}</li> \
        </ul> \
    </div>',
  data : function() {
    return {
      items : []
    }
  },
  ready : function() {
    console.log('fetch data');
    this.items = [1, 2, 3, 4, 5];
  }
},
```

在keep-alive属性下，可以在home和list之间切换currentView，home组件的ready函数只运行一次，可以看到控制台只输出了一次“fetch data”。而将keep-alive属性去除后，再次在home和list组件间切换，会发现每点击到home，控制台都会输出一次“fetch data”。

我们可以根据该特性适当地进行页面的性能优化，如果每个组件在激活时并不要求每次都实时请求数据，那使用keep-alive可以避免一些不必要的重复渲染，导致用户看到停留时间过长的空白页面。但如果每次激活组件都需要向后端请求数据的话，就不太适合使用keep-alive属性了。

Vue.js 2.0中keep-alive属性被修改为标签，例如：

```
<keep-alive>
  <component v-bind:is="view"></component>
</keep-alive>
```

6.5.3 activate钩子函数

Vue.js给组件提供了activate钩子函数，作用于动态组件切换或者静态组件初始化的过程中。activate接受一个回调函数做为参数，使用函数后组件才进行之后的渲染过程。我们将上述例子中的home组件修改为：

```
home: {
  template : '<div> \
    <p>Home</p> \
    <ul> \
        <li v-for="item in items">{{ item }}</li> \
    </ul> \
  </div>',
  data : function() {
    return {
        items : []
    }
  },
  activate : function(done) {
    var that = this;
    // 此处的setTimeout用于模拟正式业务中的ajax异步请求数据
    setTimeout(function() {
        that.items = [1, 2, 3, 4, 5];
        done();
    }, 1000);
  }
}
```

此时也可以定义两个component作为对比，并设定其中一个属性为keep-alive：

```
<component :is="currentView"></component>
```

```
<component :is="currentView" keep-alive></component>
```

可以对比出，再次激活home后，未使用keep-alive的component会延迟1s的时间才渲染出列表。

6.6 Vue.js 2.0中的变化

本小节主要说明下Vue.js 2.0 中对于组件用法及api的一些相关变化。

6.6.1 event

Vue.js 2.0中废弃了event选项，所有的自定义事件都需要通过$emit, $on, $off 函数来进行触发、监听和取消监听。另外，废弃了$dispatch和$broadcast方法。官方认为这两种方法主要依赖于组件的树形结构，而当组件结构越来越复杂后，这种事件流的形式将难以被理解，而且也并不能解决兄弟组件之间的通信问题。所以官方推荐使用集中式的事件管理机制来处理组件间的通信，而不是依赖于组件本身的结构。

官方建议可以直接使用一个空Vue实例来处理简单的事件触发机制：

```
var bus = new Vue();
bus.$emit('create', { title : 'name'});
bus.$on('create', function(data) {
  // 进行对应的操作
})
```

这样使用的话，事件的监听和触发机制就脱离了组件的结构，完全依赖于bus这个实例，在整个项目的任意地方我们都可以设置监听和触发函数。例如：

```
<div id="app">
  <comp-a></comp-a>
  <comp-b></comp-b>
</div>
var bus = new Vue();
var vm = new Vue({
  el : '#app',
  components : {
    compA : {
        template : '<div> \
            <input type="text" v-model="name" /> \
            <button @click="create">添加</button> \
          </div>',
        data : function() {
          return {
            name : ''
          }
        },
        methods : {
          create : function() {
            bus.$emit('create', { name : this.name });
            this.name = '';
          }
        }
      },
      compB : {
        template : '<ul> \
          <li v-for="item in items"> </li> \
        </ul>',
        data : function() {
          return {
            items : []
          }
        },
        // mounted 为Vue.js 2.0中新的生命周期函数
        mounted() {
          var that = this;
          bus.$on('create', function(data) {
            that.items.push(data);
          })
        }
      }
    }
  });
```

在comp-a组件中输入内容，点击“添加”即可触发create事件。在兄弟组件comp-b中则监听这个create事件，并把传入的值添加到自身的items数组中。此时的bus实例即可抽象成一个集中式的事件处理器，供所有的组件使用。

而在相对复杂的场景中，则推荐引入状态管理机制，Vuex就是这种机制与Vue.js结合的实现形式。这个会在第九章做一个详细的说明。

6.6.2 keep-alive

keep-alive不再是动态组件component标签中的属性，而成为了单独的标签。使用方式如下：

```
<keep-alive>
  <component :is="currentView"></view>
</keep-alive>
```

keep-alive也可以不和component配合使用，单独包裹多个子组件，只需要确保所有子组件只激活唯一一个即可。例如：

```
<keep-alive>
  <comp-a v-if="active"></comp-a>
  <comp-b v-else></comp-b>
</keep-alive>
```

6.6.3 slot

slot不再支持多个相同plot属性的DOM插入到对应的slot标签中，一个slot只被使用一次。下面以6.4.4节中的例子来说明：

```
// 父组件中定义了多个slot=""
<div slot="modal-header" class="modal-header">Title1</div>
<div slot="modal-header" class="modal-header">Title2</div>
// 子组件
  <slot name="modal-header"></slot>
```

在Vue.js 1.0中，父组件中的两个modal-header都会添加到slot中，而在Vue.js 2.0中，第二个modal-header会被忽略。

另外，slot标签不再保存自身的属性及样式，均由父元素或被插入的元素提供样式和属性。

6.6.4 refs

子组件索引v-ref的声明方式产生了变化，不再是一个指令了，而替换成一个子组件的一个特殊属性，例如：

```
<comp ref="first"></comp>
//Vue.js 1.0 中为
<comp v-ref="first"></first>
```

调用方式并没有发生变化，仍采用vm.$refs的方式直接访问子组件实例。

第 7 章 Vue.js常用插件

Vue.js本身只提供了数据与视图绑定及组件化等功能，如果想用它来开发一个完整的SPA（Single Page Application）应用，我们还需要使用到一些Vue.js的插件。本章主要介绍Vue-router和Vue-resouce，分别能提供路由管理和数据请求这两个功能。除此之外，还有Vue-devtools，这是一款方便查看Vue.js实例数据的chrome插件，这对我们开发和调试都有非常大的帮助。

- Vue-router
- Vue-resource
- Vue-devtools

7.1 Vue-router

Vue-router是给Vue.js提供路由管理的插件，利用hash的变化控制动态组件的切换。

引用方式如下：

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

基本用法

vue-router的基本作用就是将每个路径映射到对应的组件，并通过修改路由进行组件间的切换。常规路径规则为在当前url路径后面加上#!/path，path即为设定的前端路由路径。例如：

```
<div id="app">
 <nav class="navbar navbar-inverse">
  <div class="container">
    <div class="collapse navbar-collapse">
      <ul class="nav navbar-nav">
        <li>
          <!—使用 v-link 指令，path的值对应跳转的路径，即#!/home -->
          <a v-link="{ path : '/home'}">Home</a>
        </li>
        <li>
          <a v-link="{ path : '/list'}">List</a>
        </li>
      </ul>
    </div>
  </div>

 </nav>
 <div class="container">
 <!—路由切换组件template插入的位置 -->
  <router-view></router-view>
  </div>
 </div>
```

js代码：

```
// 创建子组件，相当于路径对应的页面
var Home = Vue.extend({
 template : '<h1>This is the home page</h1>'
});

// 创建根组件
var App = Vue.extend({})

// 创建路由器实例
var router = new VueRouter()

// 通过路由器实例定义路由规则（需要在启动应用前定义好）
// 每条路由会映射到一个组件。这个值可以是由Vue.extend创建的组件构造函数（如Home）
// 也可以直接使用组件选项对象（如'/list'中component对应的值）
router.map({
 '/home': {
   component: Home
 },
 '/list': {
   component : {
     template: '<h1>This is the List page</h1>'
   }
 }
})

// 路由器实例会创建一个Vue实例，并且挂载到第二个参数元素选择器匹配的DOM上
router.start(App, '#app')
```

嵌套路由

一般应用中的路由方式不会像上述例子那么简单，往往会出现二级导航这种情况。这时就需要使用嵌套路由这种写法。我们给上述例子添加一个Biz组件，包含一个嵌套的router-view，修改如下：

```
var Biz = Vue.extend({
  template : '<div> \
     <h1>This is the some business channel</h1> \
     <div class="container"> \
       <ul class="nav navbar-nav"> \
         <li> \
           <a v-link="{ path : \'/biz/list\'}">List</a> \
         </li> \
         <li> \
           <a v-link="{ path : \'/biz/detail\'}">Detail</a> \
         </li> \
       </ul> \
     </div> \
     <router-view></router-view> \
   </div>'
});
```

路由配置修改如下：

```
router.map({
  '/home': {
       component: Home
  },
  '/biz': {
     component : Biz,
     subRoutes : {
        '/list' : {
            component : {
              template : '<h2>This is the business list page</h2>'
            }
        },
       '/detail' : {
          component : {
            template : '<h2>This is the business detail page</h2>'
          }
       }
     }
  }
})
```

路由匹配

Vue-router在设置路由规则的时候，支持以冒号开头的动态片段。例如在设计列表分页的情况下，我们往往会在url中带入列表的页码，路由规则就可以这么设计：

```
router.map({
  '/list/:page': {
     component : {
       template: '<h1>This is the No.{{  $route.params.page }} page</h1>'
     }

  }
})
```

一条路由规则中支持包含多个动态片段，例如：

```
router.map({
  '/list/:page/:pageSize': {
     component : {
       template: '<h1>This is the No.{{ $route.params.page }} page, {{ $route.params.p
     }
  }
})
```

除了以冒号:开头的动态片段:page外，Vue-router还提供了以*号开头的全匹配片段。全匹配片段会包含所有符合的路径，而且不以'/'为间隔。例如在路由/list/:page中，规则能匹配/list/1,/list/2路径，但无法匹配/list/1/10这样的路径。而/list/*page则可以匹配/list/1, 以及/list/1/10这样的路径，不会因为'/'而中断匹配。page值也就成为整个匹配到的字符串，即1或1/10。

具名路由

在设置路由规则时，我们可以给路径名设置一个别名，方便进行路由跳转，而不需要去记住过长的全路径。例如：

```
router.map({
  '/list/:page': {
    name: 'list'
    component : {
      template: '<h1>This is the No.{{ $route.params.page }} page</h1>'
    }
  }
})
```

我们就可以使用v-link指令链接到该路径

```
<a v-link="{ name: 'list', params: { page : 1 }}">List</a>
```

路由对象

在使用Vue-router启动应用时，每个匹配的组件实例中都会被注入router的对象，称之为路由对象。在组件内部可以通过this.$route的方式进行调用。

路由对象总共包含了以下几个属性：

1. $route.path
   类型为字符串，为当前路由的绝对路径，如/list/1。
2. $route.params
   类型为对象。包含路由中动态片段和全匹配片段的键值对。如上述例子中的/list/:page路径，就可以通过this.$route.params.page的方式来获取路径上page的值。
3. $route.query
   类型为对象。包含路由中查询参数的键值对。例如/list/1?sort=createTime, 通过this.$route.query.sort即可得到createTime。
4. $route.router
   即路由实例，可以通过调用其go，replace方法进行跳转。我们在组件实例中也可以直接调用this.$router来访问路由实例。router具体的属性和api方法将在7.1.10路由实例中进行说明。
5. $route.matched
   类型为数组。包含当前匹配的路径中所有片段对应的配置参数对象。例如在/list/1?sort= createTime路径中，$route.matched值如下：

6. $route.name
   类型为字符串，即为当前路由设置的name属性。

v-link

v-link是vue-router应用中用于路径间跳转的指令，其本质是调用路由实例router本身的go函数进行跳转。该指令接受一个JavaScript表达式，而且可以直接使用组件内绑定的数据。

常见的使用方式包含以下两种：

1. 直接使用字面路径：
   `<a v-link="'home'">Home</a>` // 注意这里双引号里的home需要加上单引号，不然会变成读取组件data属性中的home值。 或者写成：`<a v-link="{ path : 'home'}">Home</a>`
2. 使用具名路径，并可以通过params或query设置路径中的动态片段或查询变量：
   `<a v-link="{ name : 'list', params: { page : 1}}">List Page 1</a>`

此外，v-link还包含其他参数选项：

1. activeClass
   类型为字符串，如果当前路径包含v-link中path的值，该元素会自动添加activeClass值的类名，默认为v-link-active。
2. exact
   类型为布尔值。在判断当前是否为活跃路径时，v-link默认的匹配方式是包容性匹配，即如果v-link中path为/list，那以/list路径为开头的所有路径均为活跃路径。而设置exact为true后，则只有当路径完全一致时才认定为活跃路径，然后添加class类名。
3. replace
   类型为布尔值。若replace值设定为true，则点击链接时执行的是router.replace()方法，而不是router.go()方法。由此产生的跳转不会留下历史记录。

4. append
   类型为布尔值。若append值设定为true，则确保链接的相对路径添加到当前路径之后。例如在路径/list下，设置链接 `<a v-link="{path: '1', append : true}">1</a>` ，点击则路径变化为/list/1；若不设置append:true, 路径变化为/1。

路由配置项

在创建路由器实例的时候，Vue-router提供了以下参数可供我们配置：

1. hashbang
   默认值为true，即只在hash模式下可用。当hashbang值为true时，所有的路径会以#!为开头。例如 `<a v-link="{ path : '/home'}">Home</a>` ，浏览器路径即为 http://hostname/#!/home
2. history
   默认值为false。设为true时会启动HTML5 history模式，利用history.pushState()和history.replaceState()来管理浏览历史记录。
3. abstract
   默认值为false。提供了一个不依赖于浏览器的历史管理工具。在一些非浏览器场景中会非常有用，例如electron（桌面软件打包工具，类似于node-webkit）或者cordova（native app打包工具，前身为phonegap）应用。
4. root
   默认值为null，仅在HTML5 history模式下可用。可设置一个应用的根路径，例如：/app。这样应用中的所有跳转路径都会默认加在这个根路径之后，例如 `<a v-link='/home'>Home</a>` ，路径即变化为/app/home。
5. linkActiveClass
   默认值为v-link-active。与v-link中的activeClasss选项类似，这里相当于是一个全局的设定。符合匹配规则的链接即会加上linkActiveClass设定的类名。
6. saveScrollPosition
   默认值为false，仅在HTML5 history模式下可用。当用户点击后退按钮时，借助HTML5 history中的popstate事件对应的state来重置页面的滚动未知。需要注意的是，当router-view设置了场景切换效果时，该属性不一定能生效。
7. transitionOnLoad
   默认值为false。在router-view中组件初次加载时是否使用过渡效果。默认情况下，组件在初次加载时会直接渲染，不使用过渡效果。
8. suppressTransitionError
   默认值为false。设定为true后，将忽略场景切换钩子函数中发生的异常。

route钩子函数

在使用Vue-router的应用中，每个路由匹配到的组件中会多出一个route选项。在这个选项中我们可以使用路由切换的钩子函数来进行一定的业务逻辑操作。以下面代码为例，介绍这些钩子函数的运行机制和触发时机。

```
var List = Vue.extend({
  template : '<h1>This is the No. page</h1>',
  route : {
    data : function(transition) {
      console.log('data');
      transition.next();
    },
    activate : function(transition) {
      console.log('activate');
      transition.next();
    },
    deactivate: function(transition) {
      console.log(deactivate);
      transition.next();
    },
    canActivate : function(transition) {
      console.log('canActivate');
      transition.next();
    },
    canDeactivate : function(transition) {
      console.log('canDeactivate');
      transition.next();
    },
    canReuse : function(transition) {
      console.log('canReuse');
      return true;
    }
  }
});
```

由上面这个例子可以看出，route提供了6个钩子函数，分别如下。

- canActivate(): 在组件创建之前被调用，验证组件是否可被创建。
- activate():在组件创建且将要加载时被调用。
- data():在activate之后被调用，用于加载和设置当前组件的数据。
- canDeactivate():在组件被移出前被调用，验证是否可被移出。
- deactivate():在组件移出时调用。
- canReuse():决定组件是否可被重用。这种场景通常发生在/list/1切换到/list/2时，如果canReuse返回值为true，则组件在切换后会略过canActivate和activate两个阶段，直接调用data钩子函数。若canReuse返回值为false，则需完整经历激活的三个钩子函数。

我们可以利用上文中的List组件设置一个路由规则：

```
router.map({
  '/home': {
     component: {
        template : '<h1>This is the home page</h1>',
     }
  },
  '/list/:page': {
     component : List
  }
})
```

在home和list之间切换，我们可以看到控制台输出结果如下：

```
canActivate
activate
data
canDeactivate
deactivate
```

在/list/1与/list/2之间切换，结果如下:

```
canActivate
activate
data
canReuse
data
```

在每个钩子函数中，都接受一个transition对象作为参数，我们称之为切换对象。主要包含以下属性和方法。

- transition.to: 将要切换到路径的路由对象（路由对象详见第7.1.6小节）。
- transition.from: 当前路径的路由对象。
- transition.next(): 可以通过调用该方法使切换过程进入下一阶段，这样也就支持了在钩子函数内部使用异步方法的情况。比如进入某个路径前我们需要校验用户是否具有某种权限，而这一般需要和后端进行数据交互来进行验证。我们只需要在异步的回调函数中执行transition.next()即可确保在获取到数据后才执行切换过程的下一阶段。
- transition.abort([reason]): 调用该方法可以终止或者拒绝此次切换。需要注意的是，在activate和deactivate中调用该方法时并不会把应用退到前一个路由状态，只有在canActivate和canDeactivate内调用才会回退。
- transition.redirect(path): 取消当前切换并重定向到另一个路由。参数接受字符串或者路由对象，并且如果不设定新的params和query的话，会保留原始transition.to的params和query。

另外，这些钩子函数在切换过程中也起到了不同的作用，我们分类说明如下。

activate/deactivate: 返回值可为Promise对象。ES6提供了原生的Promise对象，可以通过直接返回Promise.resolve(true)/Promise.reject([reason])来控制是否进行切换的下一步，或者返回return new Promise(function (resolve, reject) {resolve(true)/reject([reason]) })。

canActivate/canDeactivate: 返回值可以是同activate/deactivate一样的Promise对象，也可以是布尔值true/false，和使用切换对象transition.next()/transition.abort()效果一致。

data:data钩子在每次路由变动的时候都会被调用，特别是当组件被重用时，往往跳过activate只执行data函数，如上述例子中的/list/1切换到/list/2。所以我们经常把加载动态数据放在data钩子中执行，而且当组件从activate切换到data钩子时，会得到一个$loadingRouteData属性，默认值为true，当data函数执行完进入下一步时将切换成false。这样有助于我们做一些loading等待方面的处理，避免用户长时间得不到反馈。与其他钩子函数不同的是，我们可以在调用data函数的transition.next(data)时传入一个data对象，可以为组件的data附上相应的属性值。例如：

```
route: {
  data : function(transition) {
    transition.next({ page : transition.to.params.page })
  }
}
```

这样就可以赋值给了组件的data.page。另外，还可以通过Promise的then回调函数中的返回值来设置，例如：

```
route: {
  data : function() {
    return Promise.all([
      // 后端数据接口，需要符合Promise形式，或可通过vue-resource插件实现
      // 详情可见第7.2节中的vue-resource的相关说明
      userService.getInfo(),
      productsService.getList()
    ]).then(function(reps){
      return {
        user : reps[0],
        products : reps[1]
      }
    })
  }
}
```

路由实例属性及方法

在Vue-router启动的应用中，每个组件会被注入router实例，可以在组件内通过this.$router（或者使用路由对象$route.router）进行访问。这个router实例主要包含了一些全局的钩子函数，以及配置路由规则，进行路由切换等api。本节主要介绍路由实例的主要属性和api方法。

主要的公开属性有以下两个。

1. **router.app**
   类型为组件实例，即为路由管理的根Vue实例，是由调用router.start()传入的Vue组件构造器函数创建的。

2. **router.mode**
   类型为String，值可以为HTML5, hash或abstract，表示当前路由所采取的模式。

常见api方法如下：

1. router.start(App, el)
   启动路由应用，通过传入的组件构造器App及挂载元素el创建根组件。

2. router.stop()
   停止监听popstate和hashchange事件。调用此方法后，router.app没有被销毁，仍可以使用router.go(path)进行跳转，也可以不使用参数直接调用router.start()来重启路由。

3. router.map()
   定义路由规则的方法。包含component和subRoutes两个字段，主要用于url匹配的组件及嵌套路由。设定的路径也可以通过:冒号或*号的方式进行匹配，传

递到路由对象$route.params中。

4. router.on()
   添加一条顶级的路由配置，用法和router.map类似。例如：

```
router.on('/home', {
component : {
    template : '<h1>This is the home page.</h1>'
}
});
```

5.router.go(path)
跳转到一个新的路由。path可以是字符串也可以是包含跳转信息的对象。若使用字符串时，url直接替换成path的值。如果path不以/开头，则直接添加到当前url结尾。若path为对象，则支持以下两种格式：第一种为{ path : '…', append: true }，这种形式同直接使用字符串类似, append选项为可选，若设置成true，则确保path相对路径被添加到当前路径之后；第二种为{ name : '..', params : {}, query:{}}，name为具名路径，params和query为可选。另外，这两种格式都支持replace选项，若replace设置为true，则该跳转不产生一个新的历史记录。

vue-router 2.0 的变化

随着Vue.js升级到2.0后，Vue-router也相应做了升级。除了适配Vue.js 2.0外，vue-router 2.0对自身的使用方式，属性及钩子函数也做出了明显的改变。本节主要从以下几个方面进行说明。

1. 使用方式
   VueRouter的初始化方式、路由规则配置和启动方式均发生了变化，例如：

```
const router = new VueRouter({
// 路由规则在实例化VueRouter的时候就直接传入，而不是调用map方法再进行传递
routes : [
    { path : '/home', component: Home}
    ….
]
})
// 启动方法也发生了变化，router实例直接传入Vue.js实例中，并调用$mount方法挂载到DOM元素
const app = new Vue({
router : router
}).$mount('#app')
```

   嵌套路由的配置方法也发生了变化，改用children属性来进行标记，而且其中的path路径不需要以'/'开头，否则会认为从根路径开头。

```
const router = new VueRouter({
routes: [
{
    path: '/biz',
    component: Biz,
    children: [
    {
        path: 'list',
        component: List
        },
        {
        path: 'detail',
        component: Detail
        }
    ]
    }
]
})
```

2. 跳转方式
   路由跳转的方式也发生了变化，首先是废弃了v-link指令，采用\标签来创建a标签来定义链接。例如：

```
<router-link to="/home">
    Home
</router-link>
```

   其中的to属性和v-link所能接受的属性相同，例如{ name : 'home', params : {….}}。

   其次使用router实例方法进行跳转的api也修改成了push(), 接受的选项参数基本没有变化，例如:

```
router.push({ name : 'home', params : {…} })
```

   router.go()方法不再表示跳转，而是接受一个整型参数，作用是在history记录中向前或者后退多少步，类似 `window.history.go(n)` 。

   router实例的api方法push()、replace()、go()主要是模拟window.history下的pushState()、replaceState()和go()的使用方法来实现的，并且确保router在不同模式下（hash、history）表现的一致性。

3. 钩子函数
   Vue-router基本重新定义了自身的钩子函数，我们可以将其分为三个方面：

   - 全局钩子。
     在初始化VueRouter后直接使用router实例进行注册，包含beforeEach和afterEach两个钩子，在每个路由切换前/后调用。

```
router.beforeEach((to, from, next) => {
// to: 即将要进入的路由对象
// from: 当前正要离开的路由对象
// next: 进行下一状态，切记，一定要在结束业务逻辑后调用next函数，不然钩子函数就不
})
router.after(route=> {
// route: 进入的路由对象
})
```

- 单个路由钩子。
  这个需要在路由配置的时候直接定义，例如：

```
const router = new VueRouter({
routes: [
    {
    path: '/home',
    component: Home,
    beforeEnter: (to, from, next) => {
        // 参数和全局钩子beforeEach一致
    }
    }
]
})
```

- 组件内钩子。
  在组件内定义，例如：

```
const Home = {
template: `...`,
beforeRouteEnter (to, from, next) => {
    // 参数与全局钩子beforeEach一致
    // 切记当前钩子执行时，组件实例还没被创建，所以不能调用组件实例this
},
beforeRouteLeave (to, from, next) => {
    // 路由切换出该组件时调用，此时仍可以访问组件实例 `this`
}
}
```

4. 获取数据
   由于钩子函数的变化，在Vue.js 2.0中也就不存在使用data钩子来处理请求数据的逻辑了，可以通过监听动态路由的变化来获取数据。例如：

```
const List = {
template: '...',
watch: {
    '$route' (to, from) {
    // 对路由变化作出响应，在此处理业务逻辑
    }
}
}
```

而且在Vue.js 2.0中，我们既可以在导航完成之前获取数据，也可以在导航完成之后获取数据。在导航完成之后获取数据，是为了在获取数据期间展示一个loading状态，我们可以在组件的create()钩子函数和watch : { route : ' '} 中调用获取数据的函数，例如：

```
export default {
data () {
    return {
    ….
    }
},
created () {
    // 组件创建完后获取数据
    this.fetchData()
},
watch: {
    // 如果路由有变化，会再次执行该方法
    '$route': 'fetchData'
},
methods: {
    fetchData () {
// 调用异步请求获取数据
…..
    }
}
}
```

在导航获取之前完成数据，我们可以在beforeRouteEnter钩子中获取数据，并且只有当数据获取成功或确定有权限后才进行组件的渲染，否则就回退到路由变化前的组件状态。例如：

```
import pageSrv from './api/pages'  // 此处先模拟一个获取数据的模块

export default {
data () {
    return {
    list : []
    }
},
beforeRouteEnter (to, from, next) {
    pageSrv.get(to.params.page, (err, data) =>
    if (err) {
        next(false); // 中断当前导航
    } else {
        next(vm => {
        vm.list = data;
        })
    }
    })
},
watch: {
    $route () {
    this.list = null;
    pageSrv.get(this.$route.params.id, (err, data) => {
        if (err) {
// 处理展示错误的逻辑
        } else {
        this.list = data;
        }
    })
    }
}
}
```

5. 命名视图

   Vue-router 2.0中允许同级展示多个视图，而不是嵌套展示，可以通过给\添加name属性的方式匹配不同的组件，如果没有设置name，默认为default。例如：

```
<router-view></router-view>
<router-view name='main'></router-view>
const router = new VueRouter({
routes: [
    {
    path: '/',
    components: {  // 要注意这里的属性是components，而不是component
        default: Nav,
        main: Main
    }
    }
]
})
```

7.2 Vue-resource

Vue2.0推荐使用axios，不推荐使用Vue-resource

本节以**Vue-resource 1.0.2**版本进行说明。

引用方式

```
import VueResource from 'vue-resource';
Vue.use(VueResource);
```

使用方式

安装好Vue-resource之后，在Vue组件中，我们就可以通过this.$http或者使用全局变量Vue.http发起异步请求，例如：

```
var List = Vue.extend({
  route : {
   // vue-router中的data钩子函数，
   data : function(transition) {
     //运行这段代码需要在服务器环境中，即localhost下，直接访问文件运行这段代码会抛出异常
     this.$http
         .get('/api/list?pageNo=' + transition.to.params.page);
         .then(function(rep){
            // 成功回调函数
            transition.next({
              list : rep.data
            });
         }, function(rep) {
            // 失败回调函数
            transition.next({
              data : rep.data
            });
         });
     }
  },
  template: '<h1>This is the list page</h1>'
})
```

this.$http支持Promise模式，使用.then方法处理回调函数，接受成功/失败两个回调函数，一般会在回调函数中再调用transition.next()方法，给组件的data对象赋值，并执行组件的下一步骤。

$http的api方法和选项参数

this.$http可以直接当做函数来调用，我们以下面这个例子来对其选项进行说明：

```
this.$http({
  url : '/api/list',   // url访问路径
  method : '',     // HTTP请求方法，例如GET,POST,PUT,DELETE等
  body : {},      // request中的body数据，值可以为对象，String类型，也可以是FormData对象
  params : {},    // get方法时url上的参数，例如/api/list?page=1
  headers: {},     // 可以设置request的header属性
  timeout : 1500,  // 请求超时时长，单位为毫秒，如果设置为0的话则没有超时时长
  before : function(request) {},  //请求发出前调用的函数，可以在此对request进行修改
  progress: function(event) {},  // 上传图片、音频等文件时的进度，event对象会包含上传文件的
  credentials : boolean,  // 默认情况下，跨域请求不提供凭据(cookie、HTTP认证及客户端SSL证明
  emulateHTTP: boolean,  // 设置为true后，PUT/PATCH/DELETE请求将被修改成POST请求，并设置h
  emulateJSON : boolean  // 设置为true后，会把request body以application/x-www-form-urle
```

此外，this.$http还可以直接调用api方法，相当于提供了一些快捷方式，例如：

```
get(url, [options])
head(url, [options])
delete(url, [options])
jsonp(url, [options])
post(url, [body], [options])
put(url, [body], [options])
patch(url, [body], [options])
```

以上方法均可以采用this.$http.get(url, options)或Vue.http.get(url, options)这样类似的形式进行调用。

在发起异步请求后，我们可以采用this.$http.get(…).then()的方式处理返回值。.then()接受一个response的参数，具体的属性和方法如下。 url: response的原始url。 body：response的body数据，可以为Object，Blob，或者String类型。 headers：response的Headers对象。 ok: 布尔值，当HTTP状态码在200和299之间时为true。 status: response的HTTP状态码。 statusText: response的HTTP状态描述。 另外还包含以下三种api方法。 text(): Promise类型，把response body解析成字符串。 json(): Promise类型，把response body解析成json对象。 blob(): Promise类型，把response body解析成blob对象，即二进制文件，多用于图片、音视频等文件处理。

拦截器

拦截器主要作用于给请求添加全局功能，例如身份验证、错误处理等，在请求发送给服务器之前或服务器返回时对request/response进行拦截修改，完成业务逻辑后再传递给下一步骤。Vue-resource也提供了拦截器的具体实现方式，例如：

```
Vue.http.interceptors.push(function(request, next) {
  //修改请求
  request.method = 'POST';
  // 继续进入下一个拦截器
  next();
});
```

也可以对返回的response进行处理：

```
Vue.http.interceptors.push(function(request, next){
  request.method = 'POST';
  next(function(response) {
    // 修改response
    response.body = '...';
  });
});
```

或者直接拦截返回response，并不向后端发送请求：

```
Vue.http.interceptors.push(function(request, next) {
  // body 可自己定义，request.respondWith会将其封装成response，并赋值到response.body上
  next(request.respondWith(body, {
    status: 403,
    statusText: 'Not Allowed
  }));
});
```

$resource用法

Vue-resource提供了一种与RESTful API风格所匹配的写法，通过全局变量Vue.resource或者组件实例中的this.$resource对某个符合RESTful格式的url进行封装，使得开发者能够直接使用增删改查等基础操作，而不用自己再额外编写接口。

封装Service层

在编写SPA应用中，我们通常会把和后端做数据交互的方法封装成一个Service模块，供不同的组件进行使用。我们可以新建一个文件夹api，将Service模块集中起来，并按资源进行分类。

7.3 Vue-devtools

在开发时我们通常需要观察组件实例中的data属性的状态，方便进行调试。但一般组件实例并不会暴露在window对象上，我们无法直接访问到内部的data属性；若只通过debugger或console.log方法进行调试难免太过低效。所以Vue.js官方出了一款chrome插件Vue-devtools，它可以在chrome的开发者模式下直接查看当前页面的Vue实例的组件结构和内部属性，方便我们直接观测。

自行百度

第 8 章 Vue.js工程实例

自行百度**Vue-cli3**脚手架

第 9 章 状态管理：Vuex

自行百度**Vuex**
https://vuex.vuejs.org/zh/

Store（仓库）、State（状态）、Mutations（变更）、Actions（动作）

例子：

```
<template>
   <div id="app">
     <side></side>
     <content></content>
   </div>
</template>
```

```
   import Side from './Side.vue'
   import Content from './Content.vue'
   export default {
     components : {
       Side,
       Content
     }
   }
</script>
```

创建Side子组件，components/Side.vue。

```
<template>
   <ul class="side list-unstyled">
     <li>增加</li>
     <li>删除</li>
   </ul>
</template>
```

创建Content子组件，components/Content.vue。

```
<template>
   <div class="content">
     <div class="item" v-for="item in items">

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

创建并注入**store**

传统方式下，如果需要通过Side组件去添加Content组件中的item，只能依赖于根组件App来进行事件的监听和广播，这样既增加了耦合度，也使得Side组件和Content组件无法独立复用。 而在Vuex中，我们首先会增加Store这个概念，用于存储整个应用所需的信息，本例中将存储元素的列表。 首先，我们可以用npm先安装Vuex。

建立一个新文件vuex/store.js，代码如下：

```
import Vue from 'vue'
import Vuex from 'vuex'

Vue.use(Vuex)

// 创建一个对象来保存应用启动时的初始状态

const state = {
   items: [],  // items为元素列表,
   name : "  // 应用名称
}

// 用于更改状态的mutation函数

const mutations = {
   ….
};

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

创建好store后，需要将其注入到我们的应用中，新建文件app.js，引入Vue,Vuex及根组件App.vue。

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

new Vue({
   store,
   el: 'body',
   components: { App }
})
```

创建action及组件调用方式

action能够通过分发（dispatch），调用对应的mutation函数，来触发对store的更新。 我们在相同目录下建立vuex/actions.js。

```
export const addItem = ({ dispatch, store }, item) => {
   dispatch('ADD_ITEM', item);
}

export const deleteItem = ({ dispatch, store}) => {
   dispatch('DELETE_ITEM');
}
```

action函数也可以通过异步请求向后端获取数据，或读取store中其他的相关数据后再进行分发。例如：

```
export const getDataFromServer = ({ dispatch, store}) => {
   // 这里只是进行一个说明，你需要自己引入所需的异步请求方法
   $.ajax({
     url : '/api/data',
     success : function(data) {
       diapatch('FETCH_DATA', data);
     }
   })
}
```

在Vuex中，组件不会直接修改store对象或者自身的状态，都是通过action的方法来进行分发。下面就来修改component/Side.vue文件，使之能调用action的方法。

```
// 修改template，为增加、删除两个按键添加事件
<ul class="side list-unstyled">
   <li @click="addItem({ content : Math.random()})">增加</li>
   <li @click="deleteItem()">删除</li>
</ul>
<script>
import { addItem } from '../vuex/actions'

export default {
   data() {
     return {
     }
   },
   vuex: {
     actions: {
        addItem,
        deleteItem
     }
   }
}
</script>
```

由于之前已经注入了store，所以在子组件中，我们多了一个新的选项vuex。它可以包含一个actions属性，并将actions.js中定义的方法赋值进去，同时可以用于事件绑定。

创建mutation

action分发后就由mutation来对store进行更新。需要修改之前的vuex/store.js文件，补全在vuex/actions.js中对应的两种行为。

```
const mutations = {
   ADD_ITEM (state, item) {
     state.items.push(item);
   },

   DELETE_ITEM (state) {
     state.items.pop();
   }
};
```

对比actions.js中的方法和mutations，可以看出action在调动dispatch的时候，需要准确地传入action的名称，并且需要和mutations对象中的属性保持一致。由于动作名称往往为常量，所以我们习惯用大写的形式来命名。在大型项目中，也会单独把

动作名称集合抽象成一个模块，单独管理，例如抽象成vuex/mutation-type.js。

```
export default {
   ADD_ITEM : 'ADD_ITEM',
   DELETE_ITEM : 'DELETE_ITEM'
   …..
}
```

vuex/actions.js即可修改为：

```
import { ADD_ITEM } from './mutation-type.js'

export const addItem = ({ dispatch, store }, item) => {
   dispatch(ADD_ITEM, item);
}
```

vuex/store.js中的mutations可修改为：

```
import { ADD_ITEM } from './mutation-type.js'

const mutations = {
   [ADD_ITEM](state, item) {
   …….
}
```

组件获取state

组件中的vuex选项除了actions属性外，还有一个getters属性，里面可以定义函数，接受的参数即为vuex/store.js中定义的state对象。

修改components/Content.vue如下：

```
export default {
   vuex: {
     getters: {
       // 这里采用的ES6的写法，你可以替换成
       // items : function(state) { return state.items }
       items: state => state.items
     }
   }
}
```

这样我们在这个组件实例中就获得了state中的items数组，在template中就可以直接使用\

- \<\/li>来遍历数据。

除了在组件中直接声明getters函数外，也可以将其抽象成一个模块。例如，新建一个vuex/getters.js：

```
export function getItems(state) {
   return state.items;
}
```

components/Content.vue即可修改成：

```
import { getItems } from '../vuex/getters';
export default {
   vuex: {
     getters: {
       items: getItems
     }
   }
}
```

getters的使用并不是强制规定，只是一种最佳实践。特别是对于大型应用来说，很多组件可以共用getters方法，这样state中的值如果发生了变化，也只需要修改一个getter方法即可，而不用修改所涉及的所有组件。

小结

总结一下整体的流程

- 操作组件：单击组件按钮，调用组件中获取的action函数。
- action dispatch：action函数不会直接对store数据进行修改，而是通过dispatch的方式通知到对应的mutation。
- mutation：mutation函数则包含了对store数据的具体修改内容。
- store/state：store是包含当前state的单一对象，数据更新后，自动通知到getter函数。
- getter：getter函数从store获取组件所需的数据。
- 组件展示：组件中使用getter函数，获取新的数据，进行展示。

严格模式

Vuex.store具有严格模式，即当Vuex State在mutation函数之外的情况下被修改时，即会抛出错误。我们可以在创建实例时传入strict:true参数，即可开启严格模式：

```
const store = new Vuex.Store({
   //….
   strict : true
})
```

需要注意的是，不要在生产环境中开启严格模式。严格模式会对state树进行一个深入观察，会造成额外的性能损耗，所以可以将上述例子修改为：

```
const store = new Vuex.Store({
   //….
   strict : process.env.NODE_ENV !== 'production'
})
```

中间件

Vuex store接受middlewars选项来加载中间件，例如：

```
const store = new Vuex.Store({
   // ….
   middlewares : [myMiddleware]
})
```

myMiddleware是一个对象，可以包含设定好的钩子函数，例如：

```
const myMiddleware = {
   onInit(state) {
     // 在初始化的时候被调用，可以记录初始state
     console.log(state);
   },
   onMutation(mutation, state) {
     // 每个mutation之后都会调用
     // 每个mutation参数格式为{ type, payload}
     console.log(mutation, state);
   }
}
```

快照

可以在中间件内设置获取state的快照，用来比较mutation执行前后的state。只需在设置中间件对象的时候新增snapshot选项及onMutation钩子函数。

```
const mySnap = {
   snapshot: true,

   onMutation (mutation, nextState, prevState) {
     // nextState和prevState分别为mutation触发前和触发后对原state对象的深拷贝
   }
}
```

同严格模式一样，快照模式也建议只在开发模式下使用，处理方式与严格模式类似：

```
const store = new Vuex.store({
   //….
   middlewares : process.env.NODE_ENV !== 'production' ? [mySnap] : []
})
```

logger

为了方便调试和观察数据变化，Vuex自带了一个logger中间件，使用方法如下：

```
// 使用的vuex版本是0.82
import createLogger from 'vuex/logger';

const store = new Vuex.Store({
   middlewares : [createLogger()]
})
```

在调用action后，我们可以在控制台看到logger中间件输出的内容，记录了mutation的type和调用时间，以及state的变化过程：

createLogger有以下几个选项可供配置。

1. collapsed：默认为true，用于是否自动展开输出的mutations。
2. transformer：类型为函数，接受state为参数，用于限定在控制台输出的部分state。由于在大型应用中state通常会比较复杂，如果都直接输出到控制台会显得比较杂乱，所以可以用transformer进行控制。
3. mutationTransformer：类型为函数，接受mutation为参数，返回值即为控制台中输出的mutation。默认为{ type : '', payload : ''}，我们也可以通过设定其返回值，来对控制台的输出进行自定义。例如，我们可以设置返回值为return mutation.type，这样在控制台中仅会输出mutation.type的值，而不输出mutation.payload

createLogger使用选项的具体示例如下：

```
import createLogger from 'vuex/logger';

const logger = createLogger({
   collapsed: false,
   transformer (state) {
     return state.items
   },
   mutationTransformer (mutation) {
     return mutation.type
   }
})

const store = new Vuex.Store({
   middlewares : [logger]
})
```

表单处理

在Vuex的模式下，组件中的表单处理会稍显不同，因为表单“天然”的作用就是直接修改组件内状态，这和Vuex的action→mutation→state的修改方式显然并不符合。特别是在严格模式下。我们将第9.2节中的components/content.vue修改为：

```
<template>
   <div class="content">
     <div class="item" v-for="item in items">
       <input type="text" v-model="item.content" />
     </div>
   </div>
</template>
```

在用户输入时，就相当于直接修改state状态。而由于这个修改并不是在mutation中执行的，此时vuex就会抛出一个警告：

为了避免这种情况，也为了能够更好地跟踪state状态，我们会把表单元素绑定state的值，并在change或者blur事件中监听action行为，不推荐使用input，这样每次输入都会触发action，对性能消耗较大。例如：

```
<input :value="item.content" @change="updateContent($index, $event.target.value)" >
//  components/content.vue 的js修改为：
import { updateContent } from '../vuex/actions'

export default {
   vuex: {
     getters: {
       items: state => state.items
     },
     actions: {
       updateContent
     }
   }
}
// vuex/actions.js中增加updateContent方法
export const updateContent = ({ dispatch }, index, value) => {
   dispatch('UPDATE_CONTENT', index, value)
}
// vuex/store.js中增加mutations的UPDATE_CONTENT属性：
const mutations = {
// …..
   UPDATE_CONTENT(state, index, value) {
     // 需要注意的是，我们在本例中修改的是items数组对象中content的值
     // 如果直接写成 state.items[index].content = value，vue是无法监听到数值变化的
     // 也就无法更新视图上的content值，所以此处用$set方式更新数据
     state.items.$set(index, { content : value });
   }
};
```

当然，如果你觉得没有必要跟踪item.content的值，也可以不将此内容放入Vuex中，完全当做组件的本地状态。一般和其他组件不产生影响的状态就可以这么处理。

此外，如果希望使用状态管理，又想继续使用v-model，则可以通过Vue.js的计算属性来实现：

```
// 修改components/content.vue
<div class="content">
   <input type="text" v-model="appName">
   …..
</div>
import { updateContent, updateName } from '../vuex/actions'
export default {
   vuex: {
     getters: {
       // ….
       name : state => state.name
     },
     actions: {
       // ….
       updateName
     }
   },
   computed: {
     appName: {
       get() {
         return this.name;
       },
       set(val) {
         this.updateName(val);
       }
     }
   }
}
// 修改vuex/actions.js
export const updateName = ({ dispatch }, name) => {
   dispatch('UPDATE_NAME', name);
}
// 修改vuex/store.js
const mutations = {
   // …..
   UPDATE_NAME(state, value) {
     state.name = value;
   }
};
```

这样既能使用v-model，又对state状态进行了跟踪。如果觉得每次input事件都调用action会引起性能损耗的话，也可以使用v-model本身的lazy修饰符来降低调用频率。