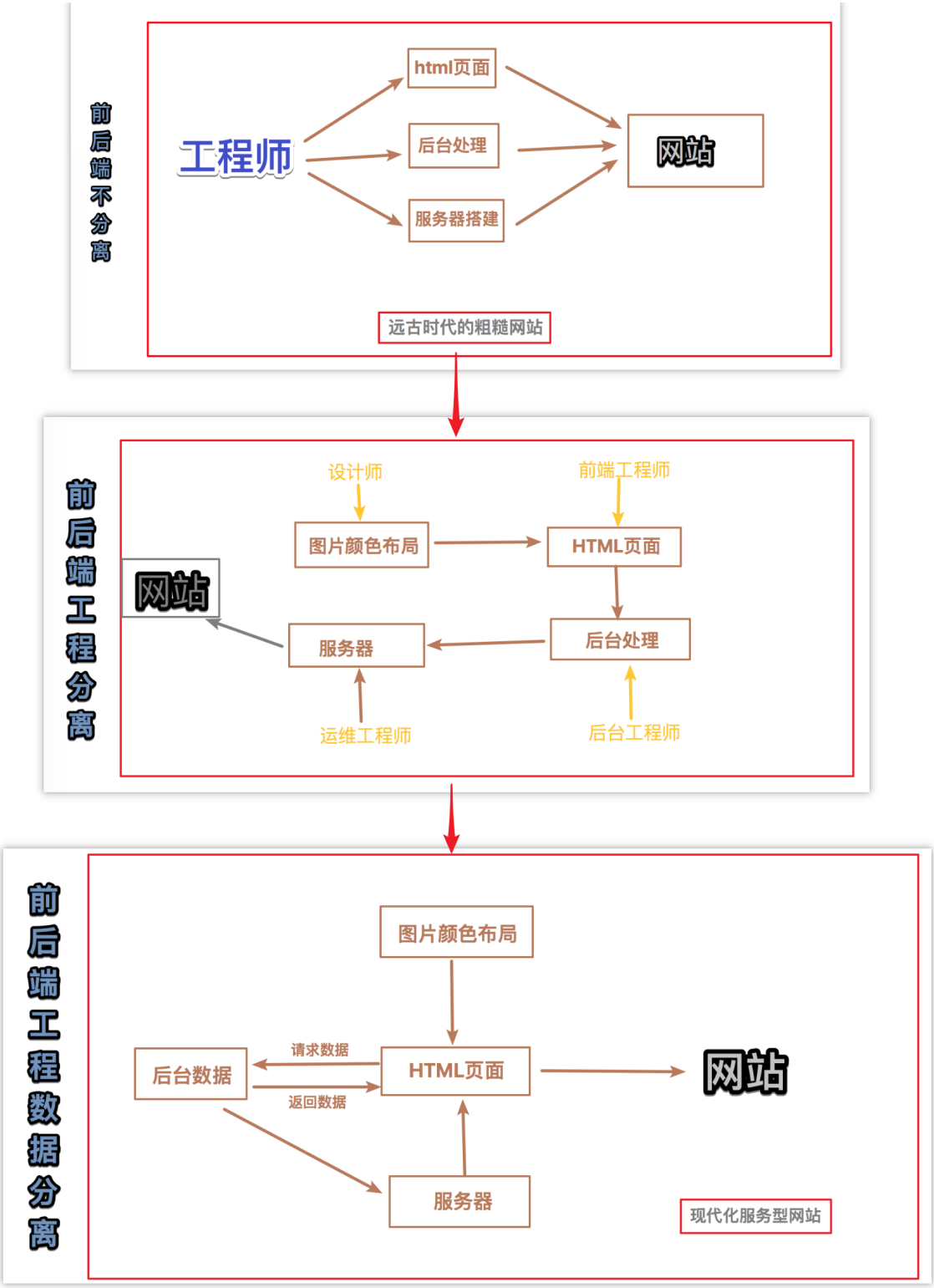
Vue.js

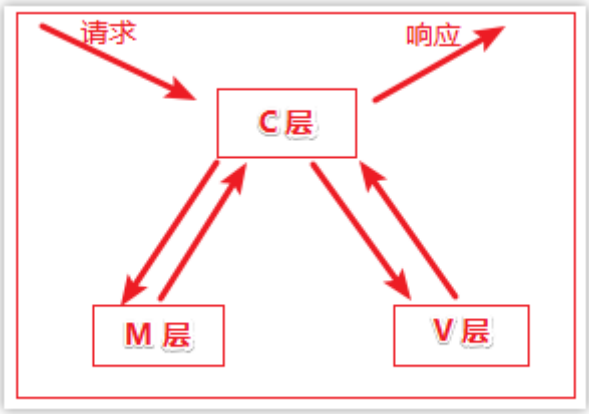
> 西岭老湿
>
> 传智播客 & 黑马程序员

第0章 Vue介绍

0.0 开发工程发展历史



通过前面的介绍，我们对目前的项目工程化有了大体了了解，那么其中，在第二阶段的工程化演进中，有一个重要的工程设计理念诞生，他就是著名的 MVC 设计模式，简单点，MVC 其实就是为了项目工程化的一种分工模式；



MVC 中的最大缺点就是单项输入输出，所有的 M 的变化及 V 层的变化，必须通过 C 层调用才能展示；

为了解决相应的问题，出现了 MVVM 的设计思想，简单理解就是实想数据层与展示层的相互调用，降低业务层面的交互逻辑；后面再进行详细介绍；

0.1 Vue 介绍

Vue (读音 /vjuː/，类似于 **view**) 是一套用于构建用户界面的 **渐进式框架。**

注意：Vue是一个框架，相对于 jq 库来说，是由本质区别的；

https://cn.vuejs.org/

Vue **不支持** IE8 及以下版本，因为 Vue 使用了 IE8 无法模拟的 ECMAScript 5 特性。但它支持所有兼容 ECMAScript 5 的浏览器。

0.2 Vue 初体验

直接下载引入：https://cn.vuejs.org/v2/guide/installation.html

CDN 引入：

```
<script src="https://cdn.jsdelivr.net/npm/vue@2.5.16/dist/vue.js"></script>
<script src="https://cdnjs.cloudflare.com/ajax/libs/vue/2.5.16/vue.js"></script>
```

CDN 加速： https://www.bootcdn.cn/

```
<body>
    <div id="div">
        {{user_name}}
    </div>
</body>

// 两种引入方式，任意选择
<script src="https://cdn.jsdelivr.net/npm/vue@2.5.16/dist/vue.js"></script>
<script src="./vue.js"></script>
```

```
<script>
    var app = new Vue({
        el:'#div',  // 设置要操作的元素
        // 要替换的额数据
        data:{
            user_name:'我是一个div'
        }
    })
</script>
```

0.3 学习Vue

基础知识 --> 项目 --> 构建工具 --> Vue其他相关技术

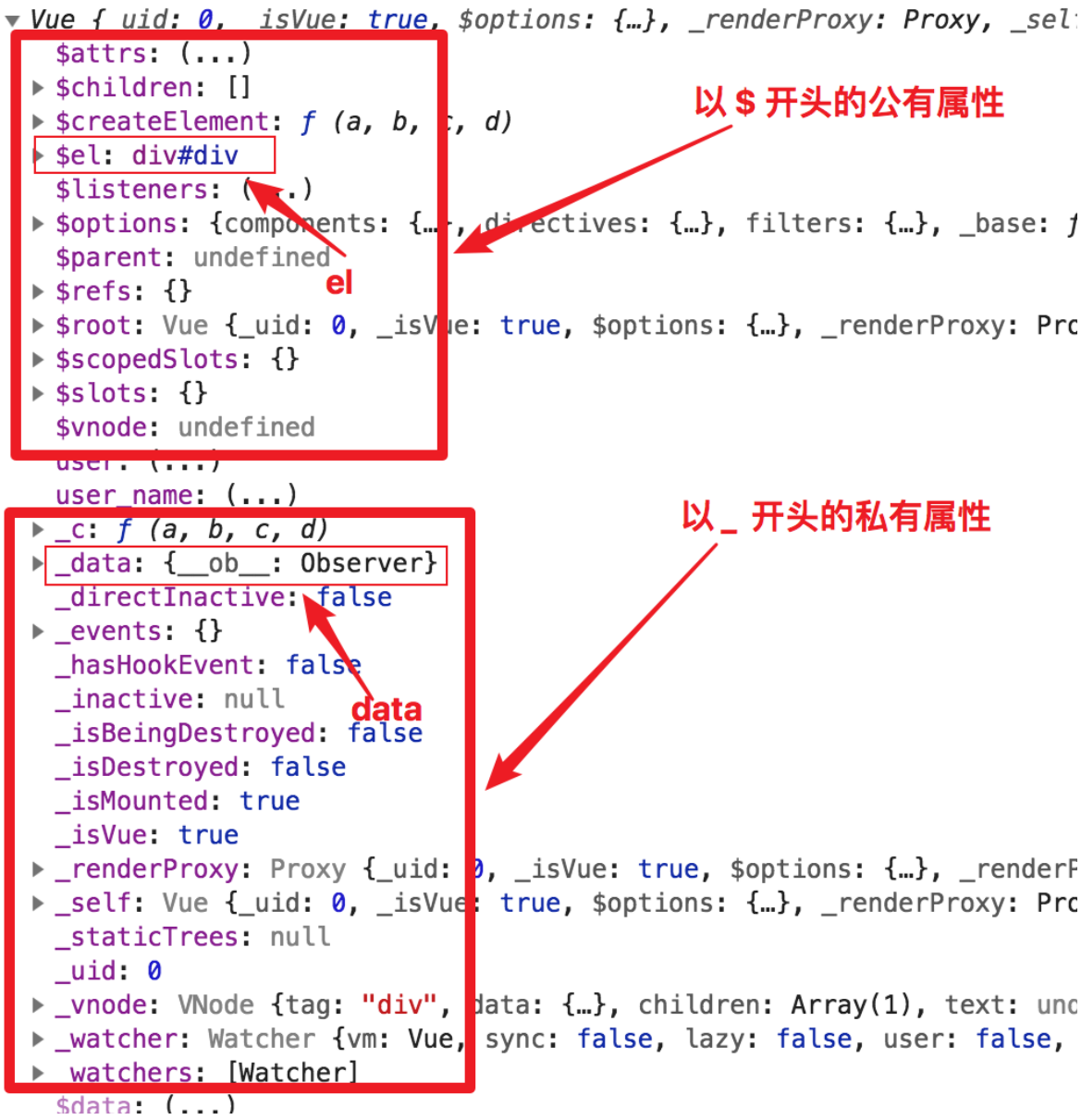
第1章 Vue 实例对象

每个 Vue 应用都是通过用 `Vue` 函数创建一个新的 **Vue 实例** 开始的：

```
var vm = new Vue({
  // 选项
})
```

```
<body>
    <div id="div">
        {{user_name}}
    </div>
</body>
<script src="./vue.js"></script>
<script>
    var app = new Vue({
        el:'#div',  // 设置要操作的元素
        // 要替换的额数据
        data:{
            user_name:'我是一个div'
        }
    })

    // 打印Vue实例对象
    console.log(app);
</script>
```



通过打印实例对象发现，其中 el 被Vue 放入了公有属性中，而data 则被放入了 私有属性中，而 data 中的数据，需要被外部使用，于是 Vue 直接将data 中的属性及属性值，直接挂载到 Vue 实例中，也就是说，data中的数据，我们可以直接使用 `app.user_name` 直接调用；

```
var app = new Vue({
    el:'#div',  // 设置要操作的元素
    // 要替换的额数据
    data:{
        user_name:'我是一个div',
        user:222222
    }
})

console.log(app.user_name);
```

第 2 章 模板语法-插值

我们在前面的代码中，使用 `{{}}` 的形式在 html 中获取实例对象对象中 data 的属性值；

这种使用 `{{}}` 获取值得方式，叫做 ***插值*** 或 ***插值表达式***；

2.1 文本

数据绑定最常见的形式就是使用“Mustache”语法 (双大括号) 的文本插值：

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

Mustache 标签将会被替代为对应数据对象上 `msg` 属性的值。无论何时，绑定的数据对象上 `msg` 属性发生了改变，插值处的内容都会更新。即便数据内容为一段 html 代码，仍然以文本内容展示

```
<body>
    <div id="div">
       文本插值 {{html_str}}
    </div>
</body>
<script>
    var app = new Vue({
        el:'#div',
        data:{
            html_str:'<h2>Vue<h2>'
        }
    })
</script>
```

浏览器渲染结果：`<div id="div">文本插值 <h2>Vue<h2></div>`

打开浏览器的 REPL 环境 输入 `app.html_str = '<s>vue</s>'`

随机浏览器渲染结果就会改变：`<div id="div">文本插值 <s>vue</s></div>`

2.2 使用 JavaScript 表达式

迄今为止，在我们的模板中，我们一直都只绑定简单的属性键值。但实际上，对于所有的数据绑定，Vue.js 都提供了完全的 JavaScript 表达式支持，但是不能使用 JS 语句；

(表达式是运算，有结果；语句就是代码，可以没有结果)

```
<body>
    <div id="div" >
       {{ un > 3 ? '大' : '小'}}
       {{ fun() }}
    </div>
</body>
<script>
    var app = new Vue({
        el:'#div',
        data:{
            un:2,
            fun:()=> {return 1+2}
        }
    })
</script>
```

第3章 模板语法-指令

指令 (Directives) 是带有 `v-` 前缀的特殊特性。指令特性的值预期是**单个 JavaScript 表达式** (`v-for` 是例外情况，稍后我们再讨论)。指令的职责是，当表达式的值改变时，将其产生的连带影响，响应式地作用于 DOM；参考 手册 、 API

```
<body>
    <div id="div" >
        <p v-if="seen">现在你看到我了</p>
    </div>
</body>
<script>
    var app = new Vue({
        el:'#div',
        data:{
            seen:false
        }
    })
</script>
```

这里，`v-if` 指令将根据表达式 `seen` 的值的真假来插入/移除 `<p>` 元素。

3.1 v-text / v-html 文本

https://cn.vuejs.org/v2/api/#v-text

https://cn.vuejs.org/v2/api/#v-html

```
<body>
    <div id="div" {{class}}>
        <p v-text="seen"></p>
        <p v-html="str_html"></p>
    </div>
</body>
<script>
    var app = new Vue({
        el:'#div',
        data:{
            seen:'<h1>Vue</h1>',
            str_html:'<h1>Vue</h1>',
            class:'dd',
        }
    })
</script>
```

<h1>Vue</h1>

Vue

> 注意：
>
> - v-text
>   - v-text和差值表达式的区别
>   - v-text 标签的指令更新整个标签中的内容(替换整个标签包括标签自身)
>   - 差值表达式，可以更新标签中局部的内容
> - v-html
>   - 可以渲染内容中的HTML标签
>   - 尽量避免使用，否则会带来危险(XSS攻击 跨站脚本攻击)

HTML 属性不能用 `{{}}` 语法

3.2 v-bind 属性绑定

https://cn.vuejs.org/v2/api/#v-bind

可以绑定标签上的任何属性。

动态绑定图片的路径

```
<img id=“app” v-bind:src="src" />
<script>
    var vm = new Vue({
        el: '#app',
        data: {
            src: '1.jpg'
        }
    });
</script>
```

绑定a标签上的id

```
<a id="app" v-bind:href="'del.php?id=' + id">删除</a>
<script>
    var vm = new Vue({
        el: '#app',
        data: {
            id: 11
        }
    });
</script>
```

绑定class

对象语法和数组语法

- 对象语法

  如果isActive为true，则返回的结果为 `<div id="app" class="active"></div>`

```
<div id="app" v-bind:class="{active: isActive}">
    hei
</div>
<script>
    var vm = new Vue({
        el: '#app',
        data: {
            isActive: true
        }
    });
</script>
```

- 数组语法

  渲染的结果： `<div id="app" class="active text-danger"></div>`

```
<div id="app" v-bind:class="[activeClass, dangerClass]">
    hei
</div>
<script>
    var vm = new Vue({
        el: '#app',
        data: {
            activeClass: 'active',
            dangerClass: 'text-danger'
        }
    });
</script>
```

绑定style

对象语法和数组语法

- 对象语法

  渲染的结果： `<div id="app" style="color: red; font-size: 40px;">hei</div>`

```
<div id="app" v-bind:style="{color: redColor, fontSize: font + 'px'}">
    hei
</div>
<script>
    var vm = new Vue({
        el: '#app',
        data: {
            redColor: 'red',
            font: 40
        }
    });
</script>
```

- 数组语法

  渲染结果： `<div id="app" style="color: red; font-size: 18px;">abc</div>`

```
<div id="app" v-bind:style="[color, fontSize]">abc</div>
<script>
    var vm = new Vue({
        el: '#app',
        data: {
            color: {
                color: 'red'
            },
            fontSize: {
                'font-size': '18px'
            }
        }
    });
</script>
```

简化语法

```
<div id="app">
    <img v-bind:src="imageSrc">
    <!-- 缩写 -->
    <img :src="imageSrc">
</div>

<script>
    var vm = new Vue({
        el: '#app',
        data: {
            imageSrc: '1.jpg',
        }
    });
</script>
```

3.3 v-model 双向数据绑定

https://cn.vuejs.org/v2/api/#v-model

单向数据绑定

```
<div id="div">
    <input type="text" :value="input_val">
</div>

<script>
    var app = new Vue({
        el: '#div',
        data: {
            input_val: 'hello world '
        }
    })
</script>
```

浏览器渲染结果：`<div id="div"><input type="text" value="hello world"></div>`

通过浏览器 REPL 环境可以进行修改 `app.input_val = 'Vue'`

浏览器渲染结果：`<div id="div"><input type="text" value="Vue"></div>`

我们通过 vue 对象修改数据可以直接影响到 DOM 元素，但是，如果直接修改 DOM 元素，却不会影响到 vue 对象的数据；我们把这种现象称为 ***单向数据绑定*** ；

双向数据绑定

```
<div id="div">
    <input type="text" v-model="input_val" >
</div>

<script>
    var app = new Vue({
        el: '#div',
        data: {
            input_val: 'hello world '
        }
    })
</script>
```

通过 v-model 指令展示表单数据，此时就完成了 ***双向数据绑定*** ；

不管 DOM 元素还是 vue 对象，数据的改变都会影响到另一个；

多行文本 / 文本域

```
<div id="div">
    <textarea v-model="inp_val"></textarea>
    <div>{{ inp_val }}</div>
</div>

<script>
    var app = new Vue({
        el: '#div',
        data: {
            inp_val: ''
        }
    })
</script>
```

绑定复选框

```
<div id="div">
    吃饭：<input type="checkbox" value="eat" v-model="checklist"><br>
    睡觉：<input type="checkbox" value="sleep" v-model="checklist"><br>
```

```
    打豆豆: <input type="checkbox" value="ddd" v-model="checklist"><br>
    {{ checklist }}
</div>

<script>
    var vm = new Vue({
       el: '#div',
        data: {
            checklist: ''
            // checklist: []
        }
    });
</script>
```

绑定单选框

```
<div id="app">
    男<input type="radio" name="sex" value="男" v-model="sex">
    女<input type="radio" name="sex" value="女" v-model="sex">
    <br>
    {{sex}}
</div>

<script>
    var vm = new Vue({
        el: '#app',
        data: {
            sex: ''
        }
    });
</script>
```

修饰符

`.lazy` - 取代 `input` 监听 `change` 事件

`.number` - 输入字符串转为有效的数字

`.trim` - 输入首尾空格过滤

```
<div id="div">
    <input type="text" v-model.lazy="input_val">
    {{input_val}}
</div>

<script>
    var app = new Vue({
        el: '#div',
        data: {
            input_val: 'hello world '
        }
    })
</script>
```

3.4 v-on 绑定事件监听

https://cn.vuejs.org/v2/api/#v-on

https://cn.vuejs.org/v2/guide/events.html

3.4.1 基本使用

```
<div id="app">
    <input type="button" value="按钮" v-on:click="cli">
</div>
<script>
    var vm = new Vue({
        el: '#app',
        data: {
            cli:function(){
                alert('123');
            }
        }
    });
</script>
```

上面的代码运行是没有问题的，但是，我们不建议这样做，因为 data 是专门提供数据的对象，事件触发需要执行的是一段代码，需要的是一个方法 (事件处理程序)；

修改代码如下：

```
<div id="app">
    <!-- 使用事件绑定的简写形式 -->
    <input type="button" value="按钮" @click="cli">
</div>
<script>
    var vm = new Vue({
        el: '#app',
        data: {},
        // 将事件处理程序写入methods对象
        methods: {
            cli: function () {
                alert('123');
            }
        }
    });
</script>
```

向事件处理器中传参

```
<div id="app">
    <!-- 直接调用传参即可 -->
    <input type="button" value="按钮" @click="cli(1,3)">
</div>
<script>
```

```
    var vm = new Vue({
        el: '#app',
        data: {},
        methods: {
            // 接受参数
            cli: function (a,b) {
                alert(a+b);
            }
        }
    });
</script>
```

而此时，如果在处理器中需要使用事件对象，则无法获取，我们可以用特殊变量 `$event` 把它传入方法

`<input type="button" value="按钮" @click="cli(1,3,$event)">`

```
methods: {
    // 接受参数
    cli: function (a,b,ev) {
        alert(a+b);
        console.log(ev);
    }
}
```

3.4.2 事件修饰符

原生 JS 代码，想要阻止浏览器的默认行为(a标签跳转、submit提交)，我们要使用事件对象的 `preventDefault()` 方法

```
<div id="app">
    <a href="http://www.qq.com" id="a">腾百万</a>
</div>
<script>
    document.getElementById('a').onclick = (ev)=>{
        // 组织浏览器的默认行为
        ev.preventDefault();
    }
</script>
```

使用修饰符 阻止浏览器的默认行为

```
<div id="app">
    <a href="http://www.qq.com" @click.prevent="cli">腾百万</a>
</div>
<script>
    var vm = new Vue({
        el: '#app',
        data: {},
        // 将事件处理程序写入methods对象
        methods: {
            cli: function () {
                alert('123');
```

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

使用修饰符绑定一次性事件

```
<div id="app">
    <a href="http://www.qq.com" @click.once="cli($event)">腾百万</a>
</div>
<script>
    var vm = new Vue({
        el: '#app',
        data: {},
        // 将事件处理程序写入methods对象
        methods: {
            cli: function (ev) {
                ev.preventDefault();
                alert('123');
            }
        }
    });
</script>
```

3.4.3 按键修饰符

绑定键盘抬起事件，但是只有`enter`键能触发此事件

```
<div id="app">
    <input type="text"  @keyup.enter="keyup">
</div>
<script>
    var vm = new Vue({
        el: '#app',
        data: {},
        methods: {
            keyup:()=>{
                console.log('111')
            }
        }
    });
</script>
```

3.4.4 系统修饰符

按住`shift`后才能触发点击事件

```
<div id="app">
    <input type="button" value="按钮" @click.shift="cli">
</div>
```

```
<script>
    var vm = new Vue({
        el: '#app',
        data: {},
        methods: {
            cli:()=>{
                console.log('111')
            }
        }
    });
</script>
```

3.4.5 鼠标修饰符

鼠标中键触发事件

```
<div id="app">
    <input type="button" value="按钮" @click.middle="cli">
</div>
<script>
    var vm = new Vue({
        el: '#app',
        data: {},
        methods: {
            cli:()=>{
                console.log('111')
            }
        }
    });
</script>
```

3.4.6 为什么在 HTML 中监听事件?

> 你可能注意到这种事件监听的方式违背了关注点分离 (separation of concern) 这个长期以来的优良传统。但不必担心，因为所有的 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，它不会导致任何维护上的困难。实际上，使用 `v-on` 有几个好处：
>
> 1. 扫一眼 HTML 模板便能轻松定位在 JavaScript 代码里对应的方法。
> 2. 因为你无须在 JavaScript 里手动绑定事件，你的 ViewModel 代码可以是非常纯粹的逻辑，和 DOM 完全解耦，更易于测试。
> 3. 当一个 ViewModel 被销毁时，所有的事件处理器都会自动被删除。你无须担心如何清理它们。

3.5 v-show 显示隐藏

https://cn.vuejs.org/v2/api/#v-show

根据表达式之真假值，切换元素的 `display` CSS 属性。

```
<div id="app">
    <p v-show="is_show">Vue</p>
</div>
<script>
    var vm = new Vue({
        el:'#app',
        data:{
            is_show:false
        },
        methods:{},
    })
</script>
```

案例：点击按钮切换隐藏显示

```
<div id="app">
    <input type="button" value="按钮" @click="isshow">
    <p v-show="is_show">Vue</p>
</div>
<script>
    var vm = new Vue({
        el:'#app',
        data:{
            is_show:false
        },
        methods:{
            isshow:function(){
                this.is_show = !this.is_show;
            }
        },
    })
</script>
```

3.6 v-if / v-else / v-else-if 条件判断

https://cn.vuejs.org/v2/api/#v-if

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

```
    var vm = new Vue({
        el: '#app',
        data: {
            type: 'F'
        },
    })
</script>
```

3.7 v-for 循环

https://cn.vuejs.org/v2/api/#v-for

```
<div id="app">
    <ul>
        <li v-for="(val,key) in arr">{{val}}---{{key}}</li>
    </ul>
    <ul>
        <li v-for="(val,key) in obj">{{val}}---{{key}}</li>
    </ul>
</div>
<script>
    var vm = new Vue({
        el: '#app',
        data: {
            arr: ['a', 'b', 'c'],
            obj: { id: 1, name: '李四' }
        },
    })
</script>
```

3.8 v-cloak

https://cn.vuejs.org/v2/api/#v-cloak

和 CSS 规则如 `[v-cloak] { display: none }` 一起用时，这个指令可以隐藏未编译的 Mustache 标签直到实例准备完毕。

```
<div id="app">
    <p>{{obj.id}}</p>
</div>
<script src="./vue.js"></script>
<script>
    setTimeout(() => {
        var vm = new Vue({
            el: '#app',
            data: {
                arr: ['a', 'b', 'c'],
                obj: { id: 1, name: '李四' }
            },
        })
    }, 2000);
```

```
</script>
```

当我们的网络受阻时，或者页面加载完毕而没有初始化得到 vue 实例时，DOM中的 `{{}}` 则会展示出来；

为了防止现象，我们可以使用 CSS 配合 v-cloak 实现获取 VUE 实例前的隐藏；

```
<style>
    [v-cloak] {
        display: none;
    }
</style>
<div id="app">
    <p v-cloak>{{obj.id}}</p>
</div>
<script src="./vue.js"></script>
<script>
    setTimeout(() => {
        var vm = new Vue({
            el: '#app',
            data: {
                obj: { id: 1, name: '李四' }
            },
        })
    }, 2000);
</script>
```

3.9 v-once

https://cn.vuejs.org/v2/api/#v-once

只渲染元素和组件**一次。**随后的重新渲染，元素/组件及其所有的子节点将被视为静态内容并跳过

```
<div id="app">
    <p v-once>{{msg}}</p>
</div>
<script>
    var vm = new Vue({
        el: '#app',
        data: {
            msg:'kkk'
        },
    })
</script>
```

第4章 TodoList 案例

上市产品： ToDoList 、奇妙清单 、滴答清单

学习练手项目： TodoMVC 、 Vue官方示例

为什么选择这样的案例：

产品功能简洁，需求明确，所需知识点丰富；实现基本功能容易，涵盖所学基础知识；而可扩展性强，完善所有功能比较复杂，所需技术众多；在学习中，可以灵活取舍；

4.1 项目初始化

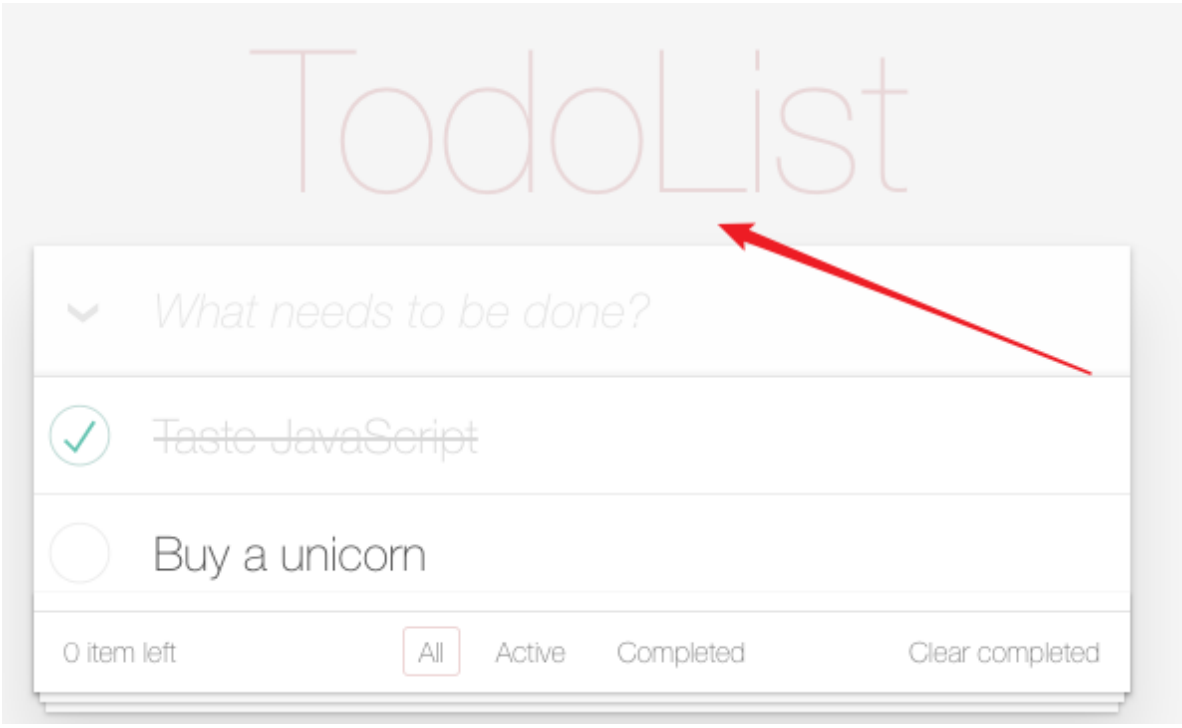
在项目目录中执行 `npm install` 命令，下载所需静态资源；将Vue.js框架代码，复制到 js 目录，在 index.html中引入 vue：`<script src="./js/vue.js"></script>`

同时 在 index.html 最下方，项目引入了app.js；而我们要写的 vuejs 代码，都放在这个文件中；

app.js

```
;(function (Vue) {
    new Vue({
        el:'#todoapp',
        data:{
            title:'TodoList'
        }
    })
})(Vue);
```

```
<header class="header">
    <!-- <h1>todos</h1> -->
    <h1>{{title}}</h1>
    <input class="new-todo" placeholde
</header>
```



4.2 数据遍历

```
const list_data = [
    {id:1,title:'吃饭',stat:true},
    {id:2,title:'睡觉',stat:false},
    {id:3,title:'打豆豆',stat:true},
]

new Vue({
    el:'#todoapp',
    data:{
        // list_data:list_data,
        list_data,// es6属性简写
    }
})
```

```
<ul class="todo-list">
    <li v-for="(val,key) in list_data">
        <div class="view">
            <input class="toggle" type="checkbox" v-model="val.stat">
            <label>{{val.title}}</label>
            <button class="destroy"></button>
        </div>
        <input class="edit" value="Rule the web">
    </li>
</ul>
```

4.3 展示无数据状态

标签及内容都是在 section footer 两个标签中的，当 list_data 中没有数据时，我们只需要隐藏这个两个标签即可：

```
<section v-if="list_data.length" class="main">
……
</section>
<footer v-if="list_data.length" class="footer">
……
</footer>
```

两个标签都有 `v-if` 判断，因此我们可以使用一个 `div` 包裹两个标签，使 `div` 隐藏即可：

```
<div v-if="list_data.length">
    <section  class="main">
    ……
    </section>
    <footer  class="footer">
    ……
    </footer>
</div>
```

如果有内容，那么 DOM 书中就会多出一个 div 标签，那么我们可以选择使用 `template` (vue中的模板标识)，有内容时，浏览器渲染不会有此节点；

```
<template v-if="list_data.length">
    <section  class="main">
    ……
    </section>
    <footer  class="footer">
    ……
    </footer>
</template>
```

4.3 添加任务

绑定 `enter` 键盘事件:

```
<input @keyup.enter="addTodo" class="new-todo" placeholder="请输入" autofocus>
```

```
new Vue({
    el:'#todoapp',
    data:{
        // list_data:list_data,
        list_data,// es6属性简写
    },
    //添加事件处理器
    methods:{
        // addTodo:function(){}
        // 简写形式
        addTodo(){
            console.log(123);
        }
    }
})
```

修改代码完成任务添加:

```
methods: {
    // 添加任务
    // addTodo:function(){}
    // 简写形式
    addTodo(ev) {
        // 获取当前触发事件的元素
        var inputs = ev.target;
        // 获取value值，去除空白后判断，如果为空，则不添加任务
        if (inputs.value.trim() == '') {
            return;
        }
        // 组装任务数据
        var todo_data = {
            id: this.list_data.length + 1 + 1,
            title: inputs.value,
            stat: false
        };
        // 将数据添加进数组
        this.list_data.push(todo_data);
        // 清空文本框内容
        inputs.value = '';
    }
}
```

4.4 任务的全选与反选

点击文本框左边的下箭头，实现全选和反选操作

为元素绑定点击事件：

```
<input @click="toggleAll" id="toggle-all" class="toggle-all" type="checkbox">
```

添加处理程序：

```
toggleAll(ev){
   // 获取点击的元素
    var inputs = ev.target;
    // console.log(inputs.checked);
    // 循环所有数据为状态重新赋值
    // 因为每个元素的选中状态都是使用 v-model 的双向数据绑定，
    // 因此 数据发生改变，状态即改变，状态改变，数据也会改变
    for(let i=0;i<this.list_data.length;i++){
        this.list_data[i].stat = inputs.checked;
    }
}
```

4.5 完成任务

如果任务完成，状态改为选中， `li` 的 `class` 属性为 `completed` 时文字有中划线；

```
<li v-for="(val,key) in list_data" v-bind:class="{completed:val.stat}">
```

4.6 删除任务

绑定点击事件，将当前索引值传入事件处理程序：

```
<button @click="removeTodo(key)" class="destroy"></button>
```

按照索引，删除相应的数据：

```
removeTodo(key){
    this.list_data.splice(key,1);
},
```

4.7 删除已完成的任务

绑定事件

```
<button @click="removeAllDone" class="clear-completed">Clear completed</button>
```

循环遍历所有数据，删除已被标记为完成的任务：

```
removeAllDone(){
    for(let i=0;i<list_data.length;i++){
        if(list_data[i].stat == true){
            this.list_data.splice(i,1);
        }
    }
}
```

循环的代码看起来很不舒服，`Array.prototype.filter()` 方法创建一个新数组，其包含通过所提供函数实现的测试的所有元素。

```
var arr = [1,4,6,2,78,23,7,3,8];
// 原始写法
// var new_arr = arr.filter(function(v){
//     // if(v>8){
//     //     return true;
//     // }
//     return v>8;
// })

// 箭头函数写法
// var new_arr = arr.filter((v)=>{
//     return v>8;
// })

// 精简写法
var new_arr = arr.filter((v)=> v>8);

console.log(new_arr);
```

修改项目代码：

```
removeAllDone(){
    // 原始循环判断用法
    // for(let i=0;i<list_data.length;i++){
    //  if(list_data[i].stat == true){
    //      this.list_data.splice(i,1);
    //  }
    // }

    // 上面是循环删除符合条件的数据
    // 下面是保留不符合条件的数据

    // 原始标准库对象方法
    // this.list_data = this.list_data.filter(function(v){
    //  if(v.stat == false){
    //      return true;
    //  }
    // })
```

```
    // 箭头函数方法
    // this.list_data = this.list_data.filter(function(v){
    //  return !v.stat;
    // })

    // 精简方法
    this.list_data = this.list_data.filter((v)=>!v.stat);
},
```

TodoList案例暂时告一段落，我们并没有将产品做完，因为我们需要用到其他知识了；

> **Vue Devtools 调试工具**
>
> 在使用 Vue 时，我们推荐在你的浏览器上安装 Vue Devtools。它允许你在一个更友好的界面中审查和调试 Vue 应用。

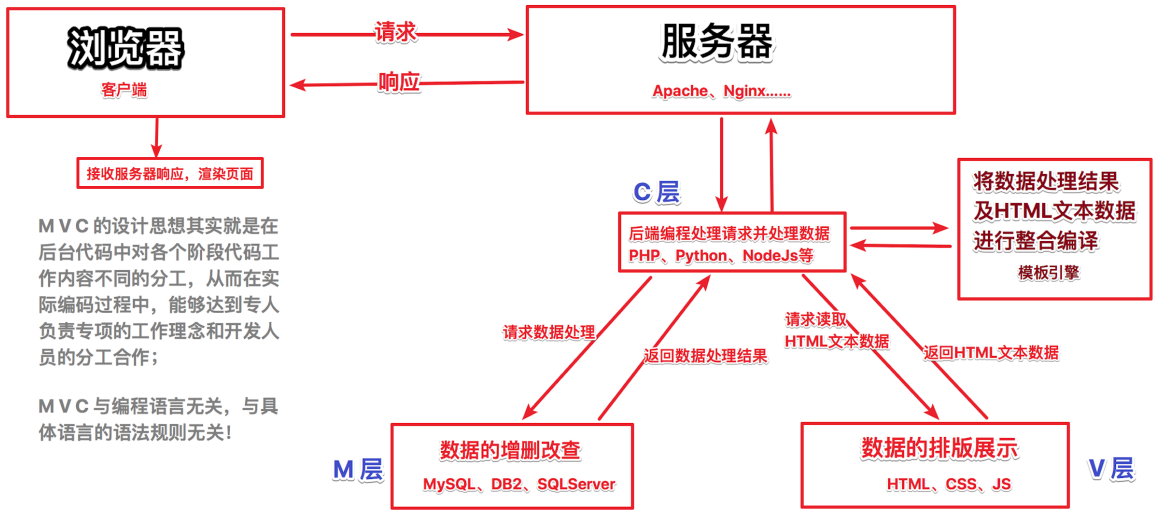
第5章 MVVM设计思想

MVC 设计思想：

M: model 数据模型层 提供数据

V: Views 视图层 渲染数据

C: controller 控制层 调用数据渲染视图



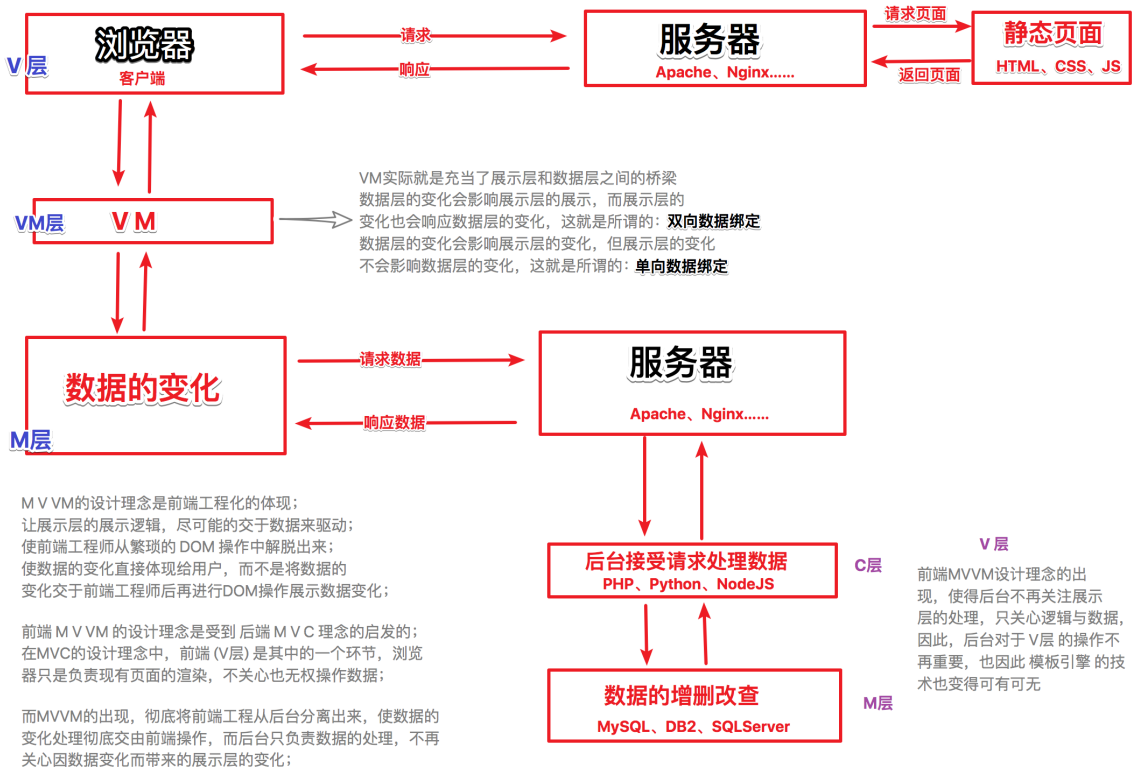
MVVM 设计思想：

M: model 数据模型层 提供数据

V: Views 视图层 渲染数据

VM：ViewsModel 视图模型层 调用数据渲染视图

由数据来驱动视图（不需要过多考虑dom操作，把重心放在VM）



第6章 其他知识点汇总

6.1 计算属性与侦听器

6.1.1 计算属性

```
<div id="div">
    <input type="text" v-model="xing">
    <input type="text" v-model="ming">
    {{xing + ming}}
</div>

<script>
    var app = new Vue({
        el: '#div',
        data: {
            xing:'',
            ming:'',
        }
    })
</script>
```

模板内的表达式非常便利，但是设计它们的初衷是用于简单运算的。在模板中放入太多的逻辑会让模板过重且难以维护。因此我们可以使用方法，来进行运算并返回数据：

```
<div id="div">
    <input type="text" v-model="xing">
```

```
    <input type="text" v-model="ming">
    {{ fullname() }}
    <!-- 一百次调用，观察时间结果-->
    {{ fullname() }}
</div>

<script>
    var app = new Vue({
        el: '#div',
        data: {
            xing:'',
            ming:'',
        },
        methods:{
            fullname(){
                return this.xing+this.ming+Date.now();
            }
        }
    })
</script>
```

注意，每次在模板中使用 `{{ fullname() }}` fullname方法就会被调用执行一次；所以，对于任何复杂逻辑，你都应当使用**计算属性**，因为计算属性，会自动缓存数据：

```
<div id="div">
    <input type="text" v-model="xing">
    <input type="text" v-model="ming">
    <br>
    {{fulln}}
    <!-- 一百次调用 -->
    {{fulln}}
</div>

<script>
    var app = new Vue({
        el: '#div',
        data: {
            xing:'',
            ming:'',
        },
        computed:{
            fulln(){
                return this.xing+this.ming+Date.now();
            }
        }
    })
</script>
```

我们可以将同一函数定义为一个方法而不是一个计算属性。两种方式的最终结果确实是完全相同的。然而，不同的是**计算属性是基于它们的依赖进行缓存的。**只在相关依赖发生改变时它们才会重新求值；多次调用，计算属性会立即返回之前的计算结果，而不必再次执行函数。

6.1.2 利用计算属性获取未完成任务个数

```
<span class="todo-count"><strong>{{getNu}}</strong> item left</span>
```

```
computed: {
    // 未完成任务个数
    getNu() {
        return (this.list_data.filter((v) => !v.stat)).length;
    }
}
```

6.1.3 使用侦听器

```
<div id="div">
    <input type="text" v-model="xing">
    <input type="text" v-model="ming">
    {{ fullname }}
</div>
<script>
    var app = new Vue({
        el: '#div',
        data: {
            xing: '',
            ming: '',
            fullname:''
        },
        // 设置侦听器
        watch: {
            // 侦听器中的方法名和要真挺的数据属性名必须一致
            // xing 发生变化，侦听器就会被执行，且将变化后的值和变化前的值传入
            xing:function(newVal,old_val){
                this.fullname = newVal+this.ming;
            },
            ming:function(newVal,oldVal){
                this.fullname = this.xing+newVal;
            }
        }
    })
</script>
```

通过上面的案例，我们基本掌握了侦听器的使用，但是我们也发现，与计算属性相比，侦听器并没有优势；也不见得好用，直观上反而比计算属性的使用更繁琐；

虽然计算属性在大多数情况下更合适，但有时也需要一个自定义的侦听器。这就是为什么 Vue 通过 `watch` 选项提供了一个更通用的方法，来响应数据的变化。当需要在数据变化时执行异步或开销较大的操作时，这个方式是最有用的。

```
<div id="div">
    <input type="text" v-model="xing">
    <input type="text" v-model="ming">
    {{ fullname }}
</div>
<script src="./jq.js"></script>
```

```
<script>
    var app = new Vue({
        el: '#div',
        data: {
            xing: '',
            ming: '',
            fullname:''
        },
        // 设置侦听器
        watch: {
            // 侦听器中的方法名和要真挺的数据属性名必须一致
            // xing 发生变化，侦听器就会被执行，且将变化后的值和变化前的值传入
            xing:function(newVal,old_val){
                // this.fullname = newVal+this.ming;
                var t = this;
                // 在侦听器中执行异步网络请求
                $.get('./xx.php',(d)=>{
                    t.fullname = d;
                })
            },
        }
    })
</script>
```

6.2 使用ref操作DOM

在学习 jq 时，我们首要任务就是学习选择的使用，因为选择可以极其方便帮助我们获取节点查找dom，因为我们要通过dom展示处理数据。而在Vue中，我们的编程理念发生了变化，变为了数据驱动dom；但有时我们因为某些情况不得不脱离数据操作dom,因此vue为我们提供了 ref 属性获取dom节点;

```
<div id="app">
    <input type="button" @click='click' value="按钮"> <br>
    <p ref="pv">123</p>
</div>
<script>
    var app = new Vue({
        el: '#app',
        methods: {
            click: function () {
                // 使用原生JS获取dom数据
                // var p = document.getElementsByTagName('p')[0].innerHTML;
                // console.log(p);

                // 使用vue ref 属性获取dom数据
                var d = this.$refs.pv.innerHTML;
                console.log(d);
            }
        }
    })
    console.log(app.$refs);
</script>
```

但是在项目开发中，尽可能不要这样做，因为从一定程度上，ref 违背的mvvm设计原则;

6.3 过滤器的使用

6.3.1 私有(局部)过滤器

定义过滤器

```
var app = new Vue({
    el: '#app',
    data:{msg:'UP'},
    //定义过滤器
    filters:{
        // 过滤器的名称及方法
        myFilters:function(val){
            return val.toLowerCase();
        }
    }
})
```

过滤器的使用：

Vue.js 允许你自定义过滤器，可被用于一些常见的文本格式化转义等操作。过滤器可以用在两个地方：**双花括号插值和 v-bind 表达式** (后者从 2.1.0+ 开始支持)。过滤器要被添加到操作值得后面，使用 管道符 `|` 分割；vue会自动将操作值，以实参的形式传入过滤器的方法中；

`{{msg|myFilters}}`

过滤敏感词汇

```
<div id="app">
    <input type="text" v-model="msg"> <br>
    {{msg|myFilters|get3}}
</div>
<script>
    var app = new Vue({
        el: '#app',
        data:{
            msg:''
        },
        //定义过滤器
        filters:{
            // 过滤器的名称及方法
            myFilters:function(val){
                return val.toLowerCase();
            },
            get3:function(val){
                // 遇到数字替换为 0
                // var reg = /\d/g;
                // return val.replace(reg,0);

                return val.replace('苍井空','***');
            }
        }
```

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

6.3.2 全局过滤器

上面的代码中，`myFilters` 及 `get3` 两个过滤器，仅在当前 vue 实例中可用；如果在代码 再次 `var app2 = new Vue()` 得到变量为 `app2` 的 vue 实例，则两个过滤器在 app2中都不可用；如果需要过滤器在所有实例对象中可用，我们需要声明 **全局过滤器**

`Vue.filter(名称,处理器)`

```
<div id="app">
    <input type="text" v-model="msg"> <br>
    {{msg|myFilters}}
</div>
<!-- 定义两个DOM节点 -->
<div id="app2">
    <input type="text" v-model="msg"> <br>
    {{msg|myFilters|get3}}
</div>
<script>
    Vue.filter('myFilters', function (val) {
        return val.toLowerCase();
    })
    // 定义两个全局过滤器
    Vue.filter('get3', function (val) {
        return val.replace('苍井空','***');
    })


    // 两个Vue 实例
    var app = new Vue({
        el: '#app',
        data: {
            msg: ''
        }
    })
    var app2 = new Vue({
        el: '#app2',
        data: {
            msg: ''
        }
    })
</script>
```

6.4 自定义指令

前面我们学过 `v-on 、v-model、v-show` 等指令，在操作 dom 时使用了 ref 属性，其实之前学过的指令也是操作dom 的一种方式，但有时，这些指令并不能满足我们的需求，因此 vue 允许我们自定义指令来操作 dom

6.4.1 全局自定义指令

```
<div id="app">
    <p v-setcolor>自定义指令的使用</p>
</div>
<script>
    // 注册一个全局自定义指令 `v-focus`
    Vue.directive('setcolor', {
        // 当被绑定的元素插入到 DOM 中时……
        inserted: function (el) {
            // 聚焦元素
            el.style.color = 'red';
        }
    })
    var app = new Vue({
        el: '#app',
    })
</script>
```

6.4.2 私有(局部)自定义指令

```
<div id="app">
    <p v-setcolor>自定义指令的使用</p>
</div>
<script>
    var app = new Vue({
        el: '#app',
        // 注册 局部(私有)指令
        directives: {
            // 定义指令名称
            setcolor: {
                // 当被绑定的元素插入到 DOM 中时……
                inserted: function (el) {
                    // 聚焦元素
                    el.style.color = 'red';
                }
            }
        }
    })
</script>
```

6.4.3 利用自定义指令使TodoList获取焦点

```
<input @keyup.enter="addTodo" v-getfocus class="new-todo" placeholder="请输入" >
```

```
// 注册 局部(私有)指令
directives: {
    // 定义指令名称
    getfocus: {
        // 当被绑定的元素插入到 DOM 中时……
        inserted: function (el) {
            // 聚焦元素
            el.focus()
        }
    }
},
```

6.4.4 为自定义指令传值

之前学习的指令中，有的指令可以传值，有的则没有，而我们自定的指令中是没有值的，如果想为自定义指令赋值，如下即可：

```
<div id="app">
    <p v-setcolor='colors'>自定义指令的使用</p>
</div>
<script>
    var app = new Vue({
        el: '#app',
        data:{
            colors:'yellow'
        },
        // 注册 局部(私有)指令
        directives: {
            // 定义指令名称
            setcolor: {
                // 自定义指令可以接受第二个参数
                inserted: function (el,val) {
                    // 第二个参数中包含了指令名称、挂载名称及数据键值
                    console.log(val);
                    // 聚焦元素
                    el.style.color = val.value;
                }
            }
        }
    })
</script>
```

6.5 过度及动画

我们可以使用v-if或者v-show控制dom元素的显示和隐藏

```
<div id="app">
    <button @click="go">显示/隐藏</button>
    <p v-show="is">pppppp1111</p>
</div>
<script>
    var app = new Vue({
```

```
        el: '#app',
        data: {
            isShow: true,
        },
        methods: {
            go() {
                this.isShow = !this.isShow;
            }
        }
    })
</script>
```

而在显示和隐藏的过程中，我们加入一些动画效果：

在进入/离开的过渡中，会有 6 个 class 切换。

1. `v-enter`：定义进入过渡的开始状态。在元素被插入之前生效，在元素被插入之后的下一帧移除。
2. `v-enter-active`：定义进入过渡生效时的状态。在整个进入过渡的阶段中应用，在元素被插入之前生效，在过渡/动画完成之后移除。这个类可以被用来定义进入过渡的过程时间，延迟和曲线函数。
3. `v-enter-to`：**2.1.8版及以上** 定义进入过渡的结束状态。在元素被插入之后下一帧生效 (与此同时 `v-enter` 被移除)，在过渡/动画完成之后移除。
4. `v-leave`：定义离开过渡的开始状态。在离开过渡被触发时立刻生效，下一帧被移除。
5. `v-leave-active`：定义离开过渡生效时的状态。在整个离开过渡的阶段中应用，在离开过渡被触发时立刻生效，在过渡/动画完成之后移除。这个类可以被用来定义离开过渡的过程时间，延迟和曲线函数。
6. `v-leave-to`：**2.1.8版及以上** 定义离开过渡的结束状态。在离开过渡被触发之后下一帧生效 (与此同时 `v-leave` 被删除)，在过渡/动画完成之后移除。

对于这些在过渡中切换的类名来说，如果你使用一个没有名字的 `<transition>`，则 `v-` 是这些类名的默认前缀。如果你使用了 `<transition name="my-transition">`，那么 `v-enter` 会替换为 `my-transition-enter`。

```
<style>
    .fade-enter-active,
    .fade-leave-active {
        transition: opacity 1s;
    }

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

    .man-enter-active,
    .man-leave-active {
        transition: opacity 4s;
    }

    .man-enter,
    .man-leave-to {
```

```
        opacity: 0;
    }
</style>

<div id="app">
    <button @click="go">显示/隐藏</button>
    <transition name="fade">
        <p v-show="isShow">pppppp1111</p>
    </transition>

    <transition name="man">
        <p v-show="isShow">pppppp222</p>
    </transition>
</div>
<script>
    var app = new Vue({
        el: '#app',
        data: {
            isShow: true,
        },
        methods: {
            go() {
                this.isShow = !this.isShow;
            }
        }
    })
</script>
```

这就是Vue中动画及过渡的基本使用方式，因为这些动画效果都需要我们自己写CSS样式，相对比较麻烦，在项目中，大多情况下，我们会借助第三方 CSS 动画库来实现，如：Animate.css ；后面项目中具体使用时，我们在进一步学习第三方 CSS 动画库的使用；

第7章 json-server与axios

一个项目从立项开始，一般都是前后端同时进行编码工作的，而此时前端需要的接口和数据后台都是无法提供的；

7.1 json-server 使用

使用全局安装： `npm install json-server -g`

json-server 会将一个json文件作为数据库来存储数据，对json数据的格式是有要求的，如data.json的内容：

```
{
  "tb1": [
    {
      "id": 1,
      "title": "标题1",
```

```
      "author": "描述信息1"
    },
    {
      "id": 2,
      "title": "标题2",
      "author": "描述信息2"
    }
  ],
  "tb2": [
    {
      "id": 1,
      "body": "some comment",
      "postId": 1
    }
  ],
  "tb3": {
    "name": "typicode"
  }
}
```

启动服务：`json-server --watch data.json`

启动成功后，提示信息如下：

```
$ json-server --watch data.json

  \{^_^}/ hi!

  Loading data.json
  Done

  Resources
  http://localhost:3000/tb1
  http://localhost:3000/tb2
  http://localhost:3000/tb3

  Home
  http://localhost:3000

  Type s + enter at any time to create a snapshot of the database
  Watching...
```

得到tb1所有的数据 GET： http://localhost:3000/tb1

根据id得到数据 GET： http://localhost:3000/tb1/2

添加一条数据 POST： http://localhost:3000/tb1

删除一条数据 DELETE： http://localhost:3000/tb1/2

模糊查找 GET： http://localhost:3000/tb1?title_like=标题

根据id修改数据 PUT： http://localhost:3000/tb1/1

> 注意：json-server 严格遵循 HTTP 请求语义进行数据处理

7.2 axios

我们在构建应用时需要访问一个 API 并展示其数据。做这件事的方法有好几种，而使用基于 **Promise** 的 HTTP 客户端 axios 则是其中非常流行的一种。

```
<script src="./axios.js"></script>
<script>
    // 获取全部数据
    axios.get('http://localhost:3000/list_data')
    .then((data)=>{
        console.log(data);
    });

    // 获取一条数据
    axios.get('http://localhost:3000/list_data/2')
    .then((data)=>{
        console.log(data);
    })

    // 添加一条数据
    axios.post('http://localhost:3000/list_data',{stat:false,title:'喝水'})
    .then((d)=>{
        console.log(d);
    }).catch(error => console.log(error))

     // 删除一条数据
     axios.delete('http://localhost:3000/list_data/4')
    .then((d)=>{
        console.log(d);
    }).catch(error => console.log(error))

     // 修改一条数据
     axios.put('http://localhost:3000/list_data/6',{title:'hhhhhh'})
    .then((d)=>{
        console.log(d);
    }).catch(error => console.log(error))
</script>
```

第8章 重构TodoList案例

8.1 启动API接口及数据

db.json:

```
{
  "list_data": [
    {
```

```
      "id": 1,
      "title": "吃饭",
      "stat": true
    },
    {
      "id": 2,
      "title": "睡觉",
      "stat": false
    },
    {
      "id": 3,
      "title": "打豆豆",
      "stat": true
    }
  ]
}
```

启动服务：`json-server --watch db.json`

8.2 获取全部任务

```
el: '#todoapp',
data: {
    // list_data:list_data,
    list_data:[]// es6属性简写
},

// 当vue实例获取到 el:'#todoapp' 自动调用执行 mounted 方法
mounted:function(){
    let url = 'http://localhost:3000/list_data';
    axios.get(url).then((backdata)=>{
        // console.log(backdata.data);
        this.list_data = backdata.data;
    })
},
```

8.3 添加任务

```
……
methods: {
    // 添加任务事件处理器
    // addTodo:function(){}
    // 简写形式
    addTodo(ev) {
        // 获取当前触发事件的元素
        var inputs = ev.target;
        // 获取value值，去除空白后判断，如果为空，则不添加任务
        if (inputs.value.trim() == '') {
            return;
        }
        // 组装任务数据
        var todo_data = {
```

```
            // 通过服务器添加数据时，不需要id值
            // id: this.list_data.length + 1 + 1,
            title: inputs.value,
            stat: false
        };
        let url = 'http://localhost:3000/list_data';
        // 将数据提交保存到服务器
        axios.post(url,todo_data).then((back_data)=>{
            let {data,status} = back_data;
            if(status == 201){
                // console.log(this.list_data);
                // 数据保存成功后，将数据添加到任务列表展示
                this.list_data.push(data);
            }
        })
        // 清空文本框
        inputs.value = '';
    },

    ……
```

8.4 删除任务

```
<button @click="removeTodo(key,val.id)" class="destroy"></button>
```

```
// 删除操作
removeTodo(key,id) {
    let url = 'http://localhost:3000/list_data/'+id;
    axios.delete(url).then((back_data)=>{
        // 结构对象
        let {data,status} = back_data;
        // console.log(back_data);
        if(status == 200){
            this.list_data.splice(key, 1);
        }
    })
},
```

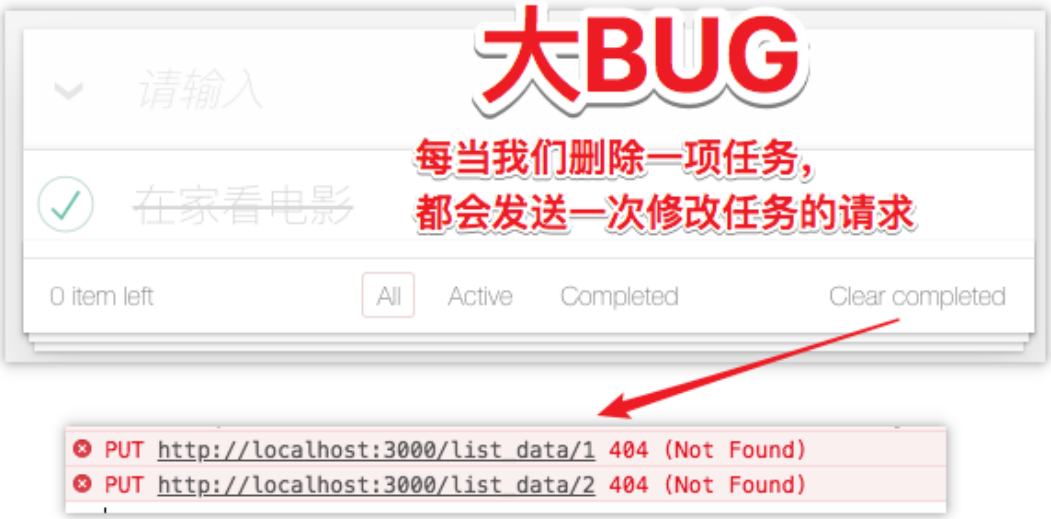
8.5 完成任务

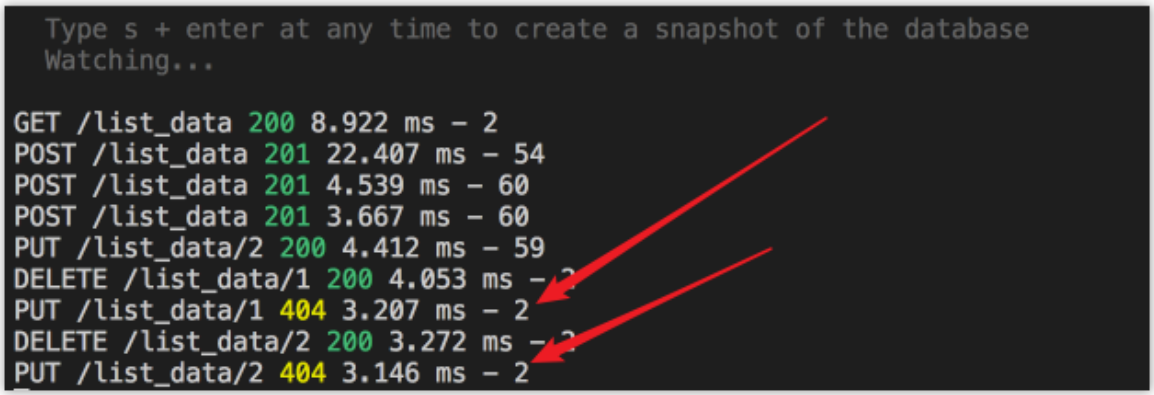
```
<li v-for="(val,key) in list_data" @click="todoDone(key,val.id)" v-bind:class="
{completed:val.stat}">
```

```
// 完成任务 事件处理器(新添加，原案例中没有)
todoDone(key,id){
    let url = 'http://localhost:3000/list_data/'+id;
    // 组装数据准备修改服务器数据
```

```
    setdata = {};
    // 注意：事件优先于浏览器渲染执行，获取当前状态
    var chestat = this.list_data[key].stat;
    // 状态取反
    setdata.stat = !chestat;
    setdata.title = this.list_data[key].title;
    // console.log(setdata);
    axios.put(url,setdata).then((backdata)=>{
        var {data,status} = backdata;
        // 如果服务器修改失败,则重新渲染DOM节点样式，改回原始状态
        // 服务器返回状态有误
        if(status != 200){
            this.list_data[key].stat = chestat;
        }
        // 如果异步执行失败失败,则重新渲染DOM节点样式，改回原始状态
    }).catch((err)=>{
        if(err){
            this.list_data[key].stat = chestat;
        }
    })
},
```

8.6 案例中的Bug





修改：`<button @click.stop="removeTodo(key,val.id)" class="destroy"></button>`

第9章 组件

https://cn.vuejs.org/v2/guide/components.html

https://cn.vuejs.org/v2/guide/components-registration.html

9.1 认识组件

组件系统是 Vue 的一个重要概念，因为它是一种抽象，允许我们使用小型、独立和通常可复用的组件构建大型应用。通常一个应用会以一棵嵌套的组件树的形式来组织：

例如，你可能会有页头、侧边栏、内容区等组件，每个组件又包含了其它的像导航链接、博文之类的组件。

9.2 基本使用

组件是可复用的 Vue 实例，且带有一个名字。把这个组件作为自定义元素来使用。组件的好处是写一次可以进行任意次数的复用。

```
<div id="app">
    <!-- 使用组件 -->
    <!-- 将组件名直接当做标签名在html代码中使用即可 -->
    <mytemp></mytemp>
    <!-- 组件可以进行任意次数的复用 -->
    <mytemp></mytemp>
</div>
<script>
    // 定义一个名为 mytemp 的新组件
    Vue.component('mytemp',{
        // template属性的值，作为组件的内容
        // vue 会把这个值替换到html中并会被浏览器渲染
        template:"<h2>我是一个组件</h2>"
    })
    var app = new Vue({
        el: '#app',
    })
</script>
```

上面代码中我们直接使用 `Vue.component()` 方法定义了组件，而这个 `mytemp` 组件可以用在所有 vue 实例中，

这种组件被称为 ***全局组件***

在具体的某个vue实例中，也可以定义组件，但是组件仅会在具体的 vue 实例中起作用，这种组件被称为 ***局部(私有)组件***

```
<div id="app">
    <!-- 使用组件 -->
    <!-- 将组件名直接当做标签名在html代码中使用即可 -->
```

```
    <mytemp></mytemp>
</div>
<div id="app2">
    <!-- 不可用 -->
    <mytemp></mytemp>
</div>
<script>
    var app = new Vue({
        el: '#app',
        // app 的私有组件，其他实例对象不可用
        components: {
            mytemp: {
                template: "<h2>我是一个组件</h2>",
            }
        }
    })
    var app2 = new Vue({
        el: '#app2',
    })
</script>
```

9.3 使用注意

组件名如果是驼峰法命名，使用组件时要将大写字母改为小写，并且在前面加上 `-`

组件中的tamplate属性必须有一个唯一的根元素，否则会报错

```
<div id="app">
    <!-- 使用组件 -->
    <!-- 将组件名直接当做标签名在html代码中使用即可 -->
    <my-temp></my-temp>
    <!-- 单标签方式使用 -->
    <my-temp/>
</div>
<div id="app2">
    <!-- 不可用 -->
    <mytemp></mytemp>
</div>
<script>
    var app = new Vue({
        el: '#app',
        // app 的私有组件，其他实例对象不可用
        components: {
            // 驼峰法命名
            myTemp: {
                // 必须有唯一的根标签，多标签报错
                template: "<div><h2>我是一个组件</h2><h3>df</h3></div>",
            }
        }
    })
    var app2 = new Vue({
        el: '#app2',
    })
```

```
</script>
```

9.4 组件的使用

CSS 代码

```
* {
    margin: 0;
    padding: 0;
}

.top {
    width: 100%;
    height: 80px;
    background-color: #ccc;
}

.left {
    margin-top: 20px;
    width: 800px;
    height: 600px;
    background-color: #ccc;
    float: left;
}

.right {
    margin-top: 20px;
    width: 400px;
    height: 600px;
    background-color: #ccc;
    float: right;
}
```

原始HTML代码

```
<div id="app">
    <div class="top">我是顶</div>
    <div class="left">我是左</div>
    <div class="right">我是右</div>
</div>
```

组件化代码

```
<div id="app">
    <tops></tops>
    <lefts></lefts>
    <rights></rights>
</div>
<script>
    var app = new Vue({
        el: '#app',
        components:{
```

```
            tops:{
                template:'<div class="top">我是顶</div>'
            },
            lefts:{
                template:'<div class="left">我是左</div>',
            },
            rights:{
                template:'<div class="right">我是右</div>'
            }
        }
    })
</script>
```

9.5 组件中的数据及方法

组件是带有名字的可复用的 **Vue 实例**，所以它们与 `new Vue` 实例对象接收相同的参数选项 `data`、`computed`、`watch`、`methods`，但 `el` 例外；

虽然组件和实例对象可以接收相同的参数选项，但在具体使用中，vue实例对象的 `data` 与组件中的 `data` 还是有差异的，在我们自己写的组件中，`data 必须是一个函数`

一个组件的 data 选项必须是一个函数，因此每个实例可以维护一份被返回的对象；

```
<div id="app">
    <my-temp></my-temp>
</div>
<script>
    var app = new Vue({
        el: '#app',
        components: {
            myTemp: {
                // 一个组件的 data 选项必须是一个函数
                data:function(){
                    // 将 数据 装入 对象 返回
                    return {msg:'我是data选项'}
                },
                // 其他选项的使用不受影响
                methods:{
                    cli(){
                        alert(123);
                    }
                },
                template: "<div @click='cli'>{{msg}}</div>",
            }
        }
    })
</script>
```

除 `data` 选项外，其他选项的使用都是一样的；

9.6 vue实例也是组件

通过`new Vue()`可以得到一个实例对象，其实这个实例对象就是一个特殊的组件，也有`template`参数，也可以当做组件来使用；

```
<div id="app">
    {{msg}}
</div>
<script>
    var app = new Vue({
        el: '#app',
        data:{msg:'数据'},
        template:'<h2>组件</h2>'
    })
</script>
```

上面的代码中直接为Vue 实例对象传入了`template`参数，那么 vue 会使用`template`中的数据替换`el`选中的整个 DOM 节点，因此`data`选项中的的数据也不会绑定，因为在绑定数据之前，整个 DOM 节点包括节点中`{{msg}}`都会被替换；如果想让数据正常绑定，我们可以在 template 数据中加入`{{msg}}`

```
<div id="app">
    {{msg}}
</div>
<script>
    var app = new Vue({
        el: '#app',
        data:{msg:'数据'},
        template:'<h2>组件{{msg}}</h2>'
    })
</script>
```

9.7 通过 Prop 向子组件传递数据 *

https://cn.vuejs.org/v2/guide/components.html

```
<div id="app">
    <mytemp></mytemp>
</div>
<script>
    var app = new Vue({
        el: '#app',
        data:{msg:'数据'},
        components:{
            mytemp:{
                template:'<h2>data:{{msg}}</h2>'
            }
        }
    })
</script>
```

运行上面的代码，我们发现，组件`mytemp`并不能获取实例中`data`的数据，这是因为组件与组件之间都拥有各自独立的作用域；

vue 在组件中提供了`props` 选项，props 接受一个在组件中自定义属性的值；

```
<div id="app">
    <mytemp cc="我是cc"></mytemp>
</div>
<script>
    var app = new Vue({
        el: '#app',
        data:{msg:'数据'},
        components:{
            mytemp:{
                template:'<h2>data:{{cc}}</h2>',
                props:['cc'],
            }
        }
    })
</script>
```

我们知道了props的用法后，怎么才能将vue实例对象中的数据传入组件中呢？我们可以借助 `v-bind` 指令来进行传值；

```
<div id="app">
    <mytemp v-bind:cc="msg" v-bind:kk="msg2"></mytemp>
</div>
<script>
    var app = new Vue({
        el: '#app',
        data:{
            msg:'数据',
            msg2:'数据二'
        },
        components:{
            mytemp:{
                template:'<h2>data:{{cc}}<br>{{kk}}</h2>',
                props:['cc','kk'],
            }
        }
    })
</script>
```

vue实例对象也是一个组件，而 `mytemp` 组件就是运行在 实例对象下面的，这时我们也会将 实例对象称为 **父组件**，将 `mytemp` 组件称为 **子组件**； 而我们上面的代码，实际上已经实现了 **父组件向子组件传递数据**的 功能；

第10章 Vue的生命周期

每个 Vue 实例在被创建时都要经过一系列的初始化过程——例如，需要设置数据监听、编译模板、将实例挂载到 DOM 并在数据变化时更新 DOM 等。同时在这个过程中也会运行一些叫做**生命周期钩子**的函数，这给了用户在不同阶段添加自己的代码的机会。

比如 `created` 钩子可以用来在一个实例被创建之后执行代码：

```
new Vue({
  data: {
    a: 1
  },
  created: function () {
    // `this` 指向 vm 实例
    console.log('a is: ' + this.a)
  }
})
// => "a is: 1"
```

也有一些其它的钩子，在实例生命周期的不同阶段被调用，如 `mounted`、`updated` 和 `destroyed`。生命周期钩子的 `this` 上下文指向调用它的 Vue 实例。

下图展示了实例的生命周期。你不需要立马弄明白所有的东西，不过随着你的不断学习和使用，它的参考价值会越来越高。

```
<div id="app">
    {{ msg }}
    <input type="text" ref="txt" v-model="msg">
</div>
<script>
    var vm = new Vue({
        el: "#app",
        data: {
            msg: 'hello vue',
            dataList: []
        },
        // 在vue对象初始化过程中执行
        beforeCreate(){
            console.log('beforeCreate');
            console.log(this.msg);// undefined
        },
        // 在vue对象初始化完成后执行
        created() {
            console.log('created');
            console.log(this.msg);//hello vue
        }
        // ……
    });
</script>
```

第11章 单页应用

11.1 单页应用

- 什么是单页应用

  单页应用(single page web application，**SPA**)，是在一个页面完成所有的业务功能，浏览器一开始会加载必需的HTML、CSS和JavaScript，之后所有的操作都在这张页面完成，这一切都由JavaScript来控制。

- 单页应用优缺点
  - 优点
    - **操作体验流畅**
    - **完全的前端组件化**
  - 缺点
    - **首次加载大量资源**(可以只加载所需部分)
    - **对搜索引擎不友好**
    - **开发难度相对较高**

优缺点都很明显，但是我们都还没尝试过就来评价，就会显得空口无凭；接下来我们先来学习制作单页应用，然后再来进行点评；

11.2 vue路由插件vue-router

https://cn.vuejs.org/v2/guide/routing.html

https://router.vuejs.org/zh/

```
<!-- 引入路由 -->
<script src="./vue.js"></script>
<script src="./vue-router.js"></script>

<div id="app">
    <ul>
        <li><a href="#/login">登录</a></li>
        <li><a href="#/register">注册</a></li>
    </ul>
    <!-- 路由中设置的组件会替换router-view标签 -->
    <router-view></router-view>
</div>
<script>
    // 1:定义路由组件
    var login = {
        template: '<h2>我是登录页面</h2>'
    }
    var register = {
        template: '<h2>注册有好礼</h2>'
    }

    // 2:获取路由对象
    var router = new VueRouter({
        // 定义路由规则
        routes: [
            // {请求的路径,componet是模板}
            { path: "/register", component: register },
            { path: "/login", component: login },
```

```
        ]
    })

    var app = new Vue({
        el: '#app',
        // ES6 属性简写
        // 3:将router对象传入Vue
        router
    })

</script>
```

上例中，在HTML中我们直接使用了 a 标签，但是这样并不好，因为官方为我们提供了 `router-link` 标签

```
<div id="app">
    <ul>
        <li><router-link to="/login">登录</router-link></li>
        <li><router-link to="/register">注册</router-link></li>

        <!-- <li><a href="#/login">登录</a></li>
        <li><a href="#/register">注册</a></li> -->

        <!-- router-link 会被解析为a标签 -->
        <!--
            不同的是，router-link在解析为a标签后，
            会自动为点击的 a 标签添加class属性
         -->
    </ul>
    <!-- 路由中设置的组件会替换router-view标签 -->
    <router-view></router-view>
</div>
```

使用 router-link 的一大好处就是，每当我们点击时，在标签内就会自动帮我们添加 class 属性，而此时，我们就可以利用 class 属性，来定义样式：

```
<style>
    .router-link-active {
        color: red;
    }
</style>
```

11.3 动态路由匹配

假设有一个用户列表，想要删除某一个用户，需要获取用户的id传入组件内，如何实现呢?

此时可以通过路由传参来实现，具体步骤如下：

1. 通过 传参，在路径上传入具体的值

```
<router-link to="/users/120">用户管理</router-link>
```

2. 路由规则中增加参数，在path最后增加 **:id**

```
{ name: 'users', path: '/users/:id', component: Users },
```

3. 在组件内部可以使用，**this.$route** 获取当前路由对象

```
var Users = {
    template: '<div>这是用户管理内容 {{ $route.params.id }}</div>',
    mounted() {
        console.log(this.$route.params.id);
    }
};
```

第12章 构建一个项目

12.0 命令行工具 (CLI)

https://cn.vuejs.org/v2/guide/installation.html#%E5%91%BD%E4%BB%A4%E8%A1%8C%E5%B7%A5%E5%85%B7-CLI

Vue 提供了一个官方的 CLI，为单页面应用 (SPA) 快速搭建繁杂的脚手架。它为现代前端工作流提供了 batteries-included 的构建设置。只需要几分钟的时间就可以运行起来并带有热重载、保存时 lint 校验，以及生产环境可用的构建版本。更多详情可查阅 Vue CLI 的文档。

12.1 初始化项目

安装 cli 命令工具：`npm install -g @vue/cli @vue/cli-init`

安装成功后，使用 `vue -V` 命令，查看版本号；

使用 `vue init webpack myapp` 构建一个名为 myapp 的项目：

Vue 依然使用询问的方式，让我们对项目有一个初始化的信息

- Project name：项目名
- Project description: 项目描述
- Author: 作者
- Vue build：
    - 第一种：配合大部分的开发人员
    - 第二种：仅仅中有runtime
- Install vue-router? 是否安装vue-router
- Use ESLint to lint your code?是否使用ESLint来验证我们的语法。
- Pick an ESLint preser:使用哪种语法规范来检查我们的代码：
    - Standard: 标准规范
    - Airbnb: 爱彼迎规范
- Set up unit test: 设置单元测试

- Setup e2e tests： 设置端对端测试
- Should we run 'npm install':要不要帮忙你下载这个项目需要的第三方包
    - 使用npm来下载
    - 使用yarn来下载

```
To get started:

  cd myapps
  npm run dev   // 使用命令启动项目

  -----
  Your application is running here: http://localhost:8080

  打开浏览器，访问 http://localhost:8080
  看到浏览器的欢迎界面，表示项目运行成功
```

12.2 项目结构介绍

```
├── build                webpack打包相关配置文件目录
├── config               webpack打包相关配置文件目录
├── node_modules          第三方包
├── src                  项目源码(主战场)
|   ├── assets            存储静态资源，例如 css、img、fonts
|   ├── components        存储所有公共组件
|   ├── router            路由
|   ├── App.vue           单页面应用程序的根组件
|   └── main.js           程序入口，负责把根组件替换到根节点
├── static               可以放一些静态资源
|   └── .gitkeep          git提交的时候空文件夹不会提交，这个文件可以让空文件夹可以提交
├── .babelrc             配置文件，es6转es5配置文件，给 babel 编译器用的
├── .editorconfig        给编辑器看的
├── .eslintignore         给eslint代码风格校验工具使用的，用来配置忽略代码风格校验的文件或
是目录
├── .eslintrc.js         给eslint代码风格校验工具使用的，用来配置代码风格校验规则
├── .gitignore           给git使用的，用来配置忽略上传的文件
├── index.html           单页面应用程序的单页
├── package.json         项目说明，用来保存依赖项等信息
├── package-lock.json     锁定第三方包的版本，以及保存包的下载地址
├── .postcssrc.js         给postcss用的，postcss类似于 less、sass 预处理器
└── README.md            项目说明文档
```

12.3 语法检查

注意： 如果我们在 构建项目时 选择了 `Use ESLint to lint your code` 那么我们在写代码时必须严格遵守 JavaScript Standard Style 代码风格的语法规则：

- **使用两个空格** – 进行缩进

- **字符串使用单引号** – 需要转义的地方除外
- **不再有冗余的变量** – 这是导致 *大量* bug 的源头!
- **无分号** – 这没什么不好。不骗你!
- 行首不要以 `(`, `[`, or ``` 开头
  - 这是省略分号时**唯一**会造成问题的地方 – *工具里已加了自动检测!*
  - 详情
- **关键字后加空格** `if (condition) { ... }`
- **函数名后加空格** `function name (arg) { ... }`
- 坚持使用全等 `===` 摒弃 `==` 一但在需要检查 `null || undefined` 时可以使用 `obj == null`。
- 一定要处理 Node.js 中错误回调传递进来的 `err` 参数。
- 使用浏览器全局变量时加上 `window` 前缀 – `document` 和 `navigator` 除外
  - 避免无意中使用到了这些命名看上去很普通的全局变量， `open`, `length`, `event` 还有 `name`。

说了那么多，看看这个遵循了 Standard 规范的示例文件 中的代码吧。或者，这里还有一大波使用了此规范的项目 代码可供参考。

> 注意: 如果你不适应这些语法规则,可以在构建项目时不使用 ESLint 的语法检查

12.4 项目代码预览

12.4.1 知识储备

严格模式

http://javascript.ruanyifeng.com/advanced/strict.html

严格模式主要有以下限制。

- 变量必须声明后再使用
- 函数的参数不能有同名属性，否则报错
- 不能使用`with`语句
- 不能对只读属性赋值，否则报错
- 不能使用前缀 0 表示八进制数，否则报错
- 不能删除不可删除的属性，否则报错
- 不能删除变量`delete prop`，会报错，只能删除属性`delete global[prop]`
- `eval`不会在它的外层作用域引入变量
- `eval`和`arguments`不能被重新赋值
- `arguments`不会自动反映函数参数的变化
- 不能使用`arguments.callee`
- 不能使用`arguments.caller`
- 禁止`this`指向全局对象
- 不能使用`fn.caller`和`fn.arguments`获取函数调用的堆栈
- 增加了保留字（比如`protected`、`static`和`interface`）

ES6模块化

http://es6.ruanyifeng.com/#docs/module

总结：

- CommonJS 模块输出的是一个值的拷贝，ES6 模块输出的是值的引用；
- CommonJS 模块是运行时加载，ES6 模块是编译时输出接口；
- ES6 的模块自动采用严格模式，不管你有没有在模块头部加上`"use strict";`；
- ES6 模块之中，顶层的`this`指向`undefined`；CommonJS 模块的顶层`this`指向当前模块；

12.4.2 代码加载执行

main.js

```
// 入口文件

// 以es6模块的方式引入 vue APP router 三个模块;
import Vue from 'vue'
import App from './App'
import router from './router'

Vue.config.productionTip = false;

/* eslint-disable no-new */
new Vue({
  // 获取节点对象
  el: '#app',
  // 引入路由
  router,
  // 本实例的私有组件
  components: { App },
  // el 与 template 在同一个实例中出现，
  // 根据生命周期的执行顺序可知，template中的内容会替换el选中的内容
  template: '<App/>'
})
```

roter/index.js

```
import Vue from 'vue'
import Router from 'vue-router'
import HelloWorld from '@/components/HelloWorld'

// Vue 中插件引入语法
// https://cn.vuejs.org/v2/guide/plugins.html
Vue.use(Router)

// ES6模块导出语法
export default new Router({
  routes: [
    // 定义一个路由规则
    {
      path: '/', // 请求路径
      name: 'HelloWorld', // 路由名称标识
      component: HelloWorld //请求此路由时，使用的组件
```

```
    }
  ]
})
```

components/HelloWorld.vue

```
export default {
  // 模块名字
  name: 'HelloWorld',
  // 组件中 data 数据必须是一个有返回值的方法
  data () {
    return {
      msg: 'Welcome to Your Vue.js App'
    }
  }
}
```

```
(main.js->template: '<App/>')替换 (index.html->div#app);

(index.html-><App/>) --> (components: { App })

( components: { App }) --> (import App from './App' -> src/App.vue)

(App.vue -> <router-view/> -> 路由组件) --> (main.js-> router)
========此项决定了页面展示那个组件内容 ========

({path: '/',name: 'HelloWorld', component: HelloWorld }) --> (import HelloWorld
from '@/components/HelloWorld')

(src/components/HelloWorld.vue) --> <router-view/>
```

12.5 添加自己的路由组件

修改 router/index.js， 添加自己的路由

```
import Vue from 'vue'
import Router from 'vue-router'
import HelloWorld from '@/components/HelloWorld'
// 引入(导入) 组件
import MyRouter from '@/components/MyRouter'

Vue.use(Router)

// ES6模块导出语法
```

```
export default new Router({
  routes: [
    {path: '/',name: 'HelloWorld', component: HelloWorld },
    // 添加自己的路由及组件
    {
      path:'/myrouter',
      name:'MyRouter',
      component:MyRouter
    }
  ]
})
```

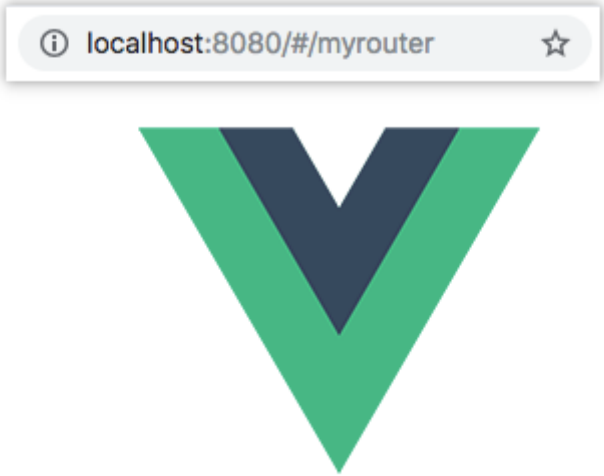
在 components 文件夹中添加 MyRouter.vue 文件，写自己的组件代码：

```
<template>
  <div class="mypage">
    {{mydatas}}
  </div>
</template>

<script>
  // 模块化导出
  export default {
    data(){
      return {mydatas:'lksadjflks'}
    }
  }
</script>

<style>
  .mypage{
    width: 200px;
    height: 50px;
    background: pink
  }
</style>
```

浏览器渲染效果如下：



lksadjflks