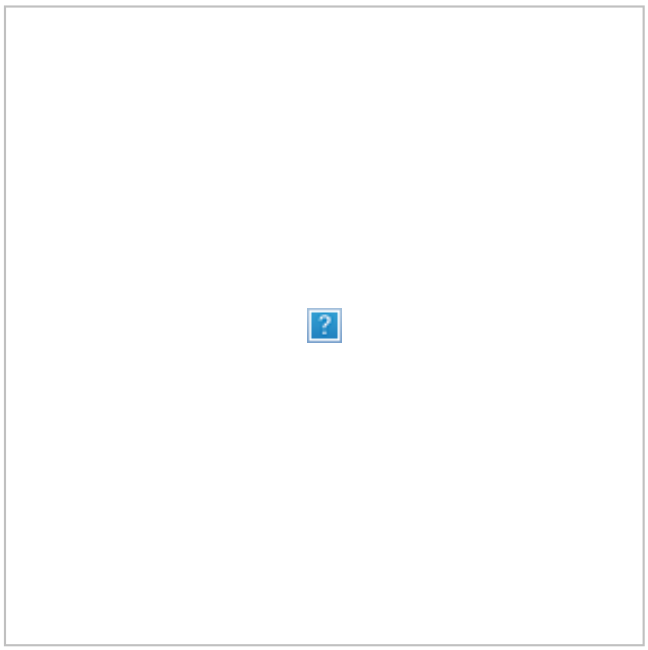
vue第一节

vue是什么

简介：vue.js(读音类似view)是一套构建用户界面的渐进式框架.发布时间为2014年2月份。作者为尤雨溪(微博尤小右，知乎尤雨溪)。最初的定位并不是框架而是视图层的库，而现如今vue的生态圈蓬勃发展，vue-router,vue-resourece,vuex等第三方插件的推出，已经成长为一个框架了.



出现的原因：

浏览器升级

进年来旧浏览器逐渐被淘汰，ie8在2015年为22%，16年为16%，到17年近三个月已经跌到了12%。xx.而像jquery主要处理的还是以pc的兼容操作为主

vue不兼容ie8及以下浏览器，可以使用es5的诸多特性，让它在桌面和移动端都能大显身手

前端越来越复杂

前端交互也越来越多，功能越来越复杂. 想想你用的购物网站，视频分享平台，音乐互动社区，打车出行等

架构升级

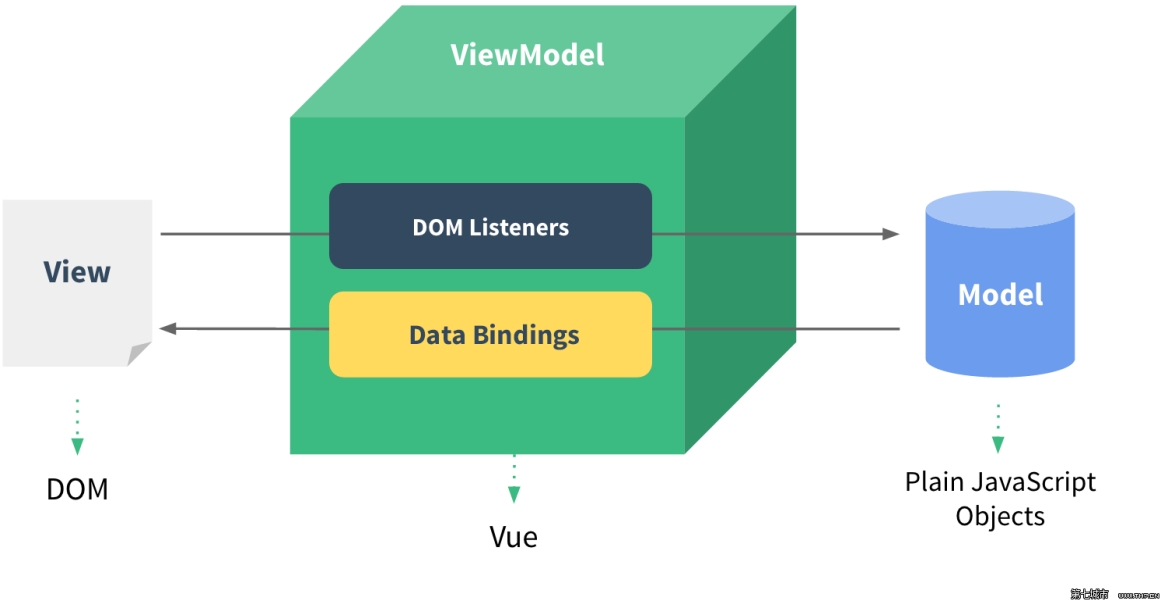
架构从传统后台mvc向ajax+restAPI+前端的MV**(mvc,mvp,mvvm)迁移，而vue是mvvm框架

mvvm框架

mvvm框架主要包含3个部分(view、model、viewModel)

- model 就是数据-javascript对象

- view 就是视图-对应的就是dom
- viewModel-就是连接视图和数据的中间件-通讯



在mvvm的架构中是不允许数据和视图直接通讯的，只能通过viewModel来通讯，而viewModel就是定义一个Observer观察者，当数据发生变化，viewModel能够观察到数据的这个变化，并对视图对应的内容进行更新,而用户操作视图，也能监听到视图的变化，通知数据发生改变.

使用的场景：

- 针对具有复杂交互逻辑的前端应用
- 提供基础的架构抽象
- 通过ajax数据持久化，保证前端用户体验，特别是在移动端，刷新或者跳转页面的成本太高（需要加载资源）所以在移动端不管是H5的邀请函小项目还是webapp的重量级项目都会使用SPA（单页应用）
- 不需要手动更改页面dom

使用mvvm架构的框架：

- angular.js
- react.js
- vue.js
- 微信小程序

也是目前前端最火的三个框架

vuejs的特点

他是一个轻量级的mvvm框架，压缩后只有20多k.(ps:在移动端项目中很多公司对文件大小都有限制，比如京东的h5页面首屏加载大小就不能超过400k)

数据驱动+组件化的前端开发

github超过49k+的,超过jquery的44k,社区完善.使用量众多.

对比angular和react

1.vue更轻量：

- vue20k+
- angular56k+
- react44k+

在移动端更适合

2.vuejs更易上手，学习曲线平稳.文档丰富.

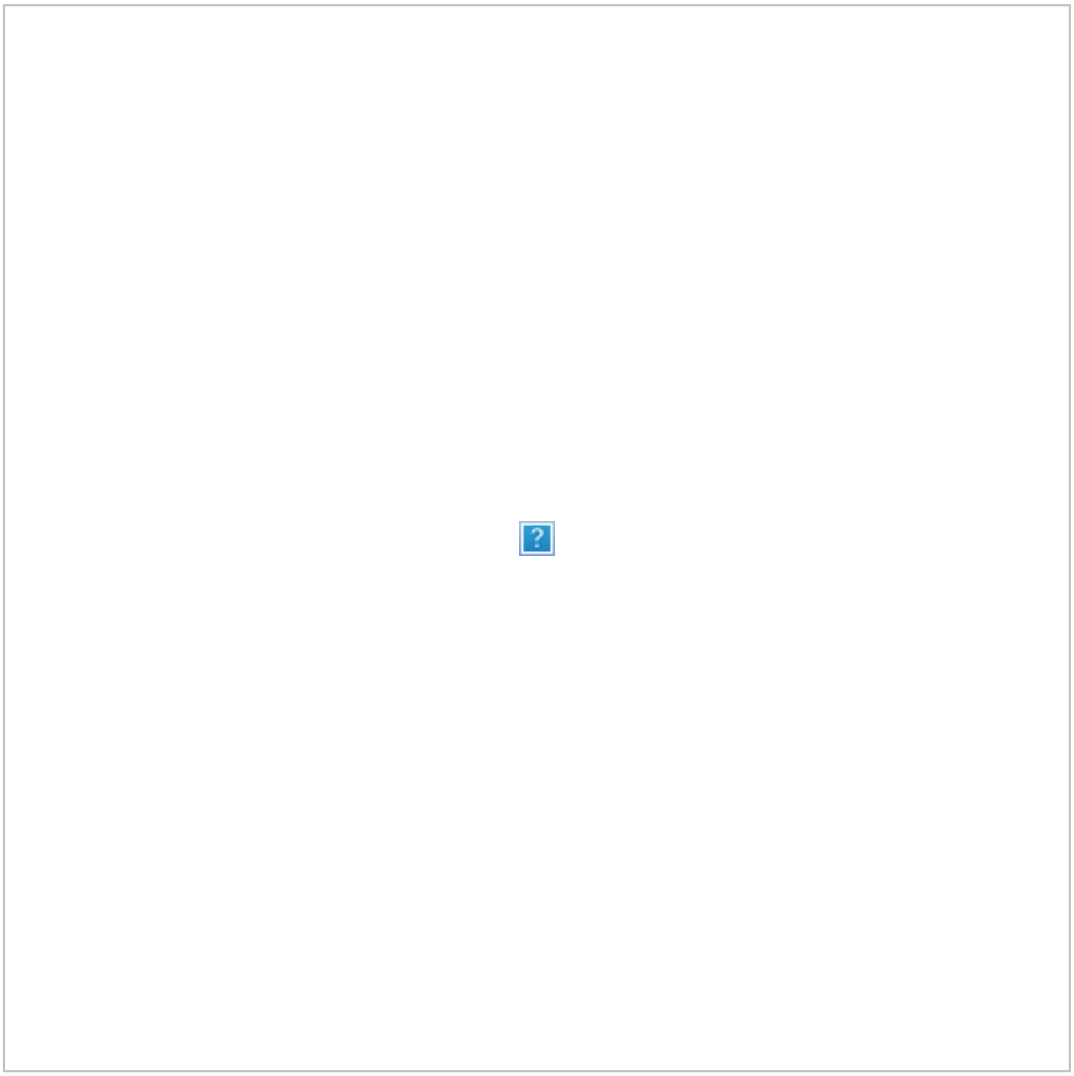
比如angular是由一帮做java后台的搞出来的，很多的后台开发中的概念，目前的最新版本4.x，开发语言推荐的是typescript(javascript超集).你找不到一个完善的关于angularAPI的中文文档，学习曲线非常陡峭

而react有一个jsx语法.对很多同学来说也是一个非常大的挑战，开发工具的支持也很弱.最主要的学习react还会附赠react全家桶，比如redux，react-router等.虽然vue中也有类似的工具但使用起来要方便得多.

3. 吸取了两家之长，借鉴了angular的指令和react的组件化，同时又开发了独特又好用的功能，比如计算属性

vue的关键字：数据驱动、组件化

数据驱动：



组件化：

- 页面上每个独立的可视/可交互的区域视为一个组件
- 每个组件对应一个工程目录，组件所需要的各种资源在这个目录下就近维护
- 页面不过是组件的容器，组件可以嵌套自由组合形成完整的页面

实例：notes项目

vue的文档

官网地址 api地址

vue的使用

环境搭建

兼容性：vue不支持ie8及以下的浏览器

下载：开发版本 生产版本

使用npm安装最新稳定版：`$ npm install vue`

在页面中引入vue.js远程文件或者本地文件

```
<script src="https://unpkg.com/vue/dist/vue.js"></script>
```

数据绑定

vue允许采用简洁的模板语法来声明式的将数据渲染进 DOM

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Document</title>
    <script type="text/javascript" src="js/vue.js"></script>
</head>
<body>
    <div id="app">
        <!-- 允许采用简洁的模板语法来声明式的将数据渲染进 DOM -->
        {{message}}
    </div>
</body>
<script type="text/javascript">
    var app = new Vue({
        el:"#app",
        data:{
            message:'hello Vue!'
        }
    });
</script>
</html>
```

也可以通过绑定事件来更改数据从而影响视图：

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Document</title>
    <script type="text/javascript" src="js/vue.js"></script>
</head>
<body>
    <div id="app">
        <!-- 允许采用简洁的模板语法来声明式的将数据渲染进 DOM -->
        {{message}}
        <button v-on:click="changeMsg">更改message</button>
    </div>
</body>
<script type="text/javascript">
    var app = new Vue({
        el:"#app",
        data:{
            message:'hello Vue!'
        },
        methods:{
            changeMsg(){
                this.message = "hello tangcaiye";
            }
        }
    });
</script>
</html>
```

除了文本插值，还可以绑定属性，只是在属性中绑定得需要使用v-bind

```
<span v-bind:title="message">
  查看你加载这个页面的时间可以通过鼠标移入到这段文字上
</span>
```

当然当你改变message这个数据的时候title中的message也会发生改变

条件与循环

可以通过v-if指令去控制切换一个元素的显示隐藏

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Document</title>
    <style type="text/css">
    .fa-enter-active, .fa-leave-active {
      transition: opacity .5s
    }
    .fa-enter, .fa-leave-active {
      opacity: 0
    }
    </style>
    <script type="text/javascript" src="js/vue.js"></script>
</head>
<body>
    <div id="app">
        <!--添加过渡-->
         <transition name="fa">
             <span v-if="seen">我是一段文字内容</span>
         </transition>
    </div>
</body>
<script type="text/javascript">
    var app = new Vue({
         el:"#app",
         data:{
             seen:true
         }
    });
</script>
</html>
```

可以通过v-for循环来渲染一个项目列表

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Document</title>
    <script type="text/javascript" src="js/vue.js"></script>
</head>
<body>
    <div id="app">
        <ol>
            <li v-for="item in todos">
                {{item.text}}
            </li>
        </ol>
    </div>
</body>
<script type="text/javascript">
    var app = new Vue({
        el:"#app",
        data:{
            todos:[
                {text:"学习javascript"},
                {text:"学习vue"},
                {text:"学习给小孩换尿片"}
            ]
        }
    });

    document.onclick = function (){
        app.todos.push({
            text:"打游戏"
        });
    }
</script>
</html>
```

双向数据绑定

Vue 还提供了 `v-model` 指令，它能轻松实现表单输入和应用状态之间的双向绑定

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Document</title>
    <style type="text/css">
        .a-enter-active, .a-leave-active {
            transition: opacity .5s
        }
        .a-enter, .a-leave-active {
            opacity: 0
        }
    </style>
    <script type="text/javascript" src="vue.js"></script>
</head>
<body>
    <div id="app">
        {{message}}
        <br>
        <!--<input type="text" v-model="message">-->
        <input type="text" v-bind:value="message">
    </div>
</body>
<script type="text/javascript">
    var vm = new Vue({
        el:"#app",
        data:{
            message:'我是你好'
        }
    });
</script>
</html>
```

组件化的应用构建

组件系统是 Vue 的另一个重要概念，因为它是一种抽象，允许我们使用小型、自包含和通常可复用的组件构建大型应用。仔细想想，几乎任意类型的应用界面都可以抽象为一个组件树：

在 Vue 里，一个组件本质上是一个拥有预定义选项的一个 Vue 实例，在 Vue 中注册组件很简单：

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Document</title>
    <script type="text/javascript" src="js/vue.js"></script>
</head>
<body>
    <div id="app">
        <p>{{message}}</p>
        <ol>
            <todo-item></todo-item>
        </ol>
    </div>

</body>
<script type="text/javascript">
    //这是在全局定义了一个叫todo-item的组件
    Vue.component("todo-item",{
        template:'<li>这是一个代办事项</li>'
    });
    var app = new Vue({
        el:"#app",
        data:{
            message:"hello vue!"
        }
    });


</script>
</html>
```

但是这样会为每个待办项渲染同样的文本，这看起来并不炫酷，我们应该能将数据从父作用域传到子组件。让我们来修改一下组件的定义，使之能够接受一个属性

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Document</title>
    <script type="text/javascript" src="js/vue.js"></script>
</head>
<body>
    <div id="app">
        <ol>
            <!-- 现在我们为每个todo-item提供待办项对象    -->
            <!-- 待办项对象是变量，即其内容可以是动态的 -->
            <todo-item v-for="item in groceryList" v-bind:todo="item"></todo-item>
        </ol>
    </div>

</body>
<script type="text/javascript">
    Vue.component("todo-item",{
        // todo-item 组件现在接受一个
        // "prop"，类似于一个自定义属性
        // 这个属性名为 todo。
        props:['todo'],
        template:'<li>食物都有：{{todo.text}}</li>'
    });
    var app = new Vue({
        el:"#app",
        data:{
            groceryList:[
                {text:'蔬菜'},
                {text:'起司'},
                {text:'米饭'}
            ]
        }
    });
</script>
</html>
```

vue实例

构造器

每个 Vue.js 应用都是通过构造函数 `Vue` 创建一个 **Vue 的根实例** 启动的：

```
var vm = new Vue({
  // 选项
})
```

虽然没有完全遵循 MVVM 模式， Vue 的设计无疑受到了它的启发。因此在文档中经常会使用 `vm` (ViewModel 的简称) 这个变量名表示 Vue 实例.

在实例化 Vue 时，需要传入一个**选项对象**，它可以包含数据、模板、挂载元素、方法、生命周期钩子等选项。全部的选项可以在 API 文档中查看。

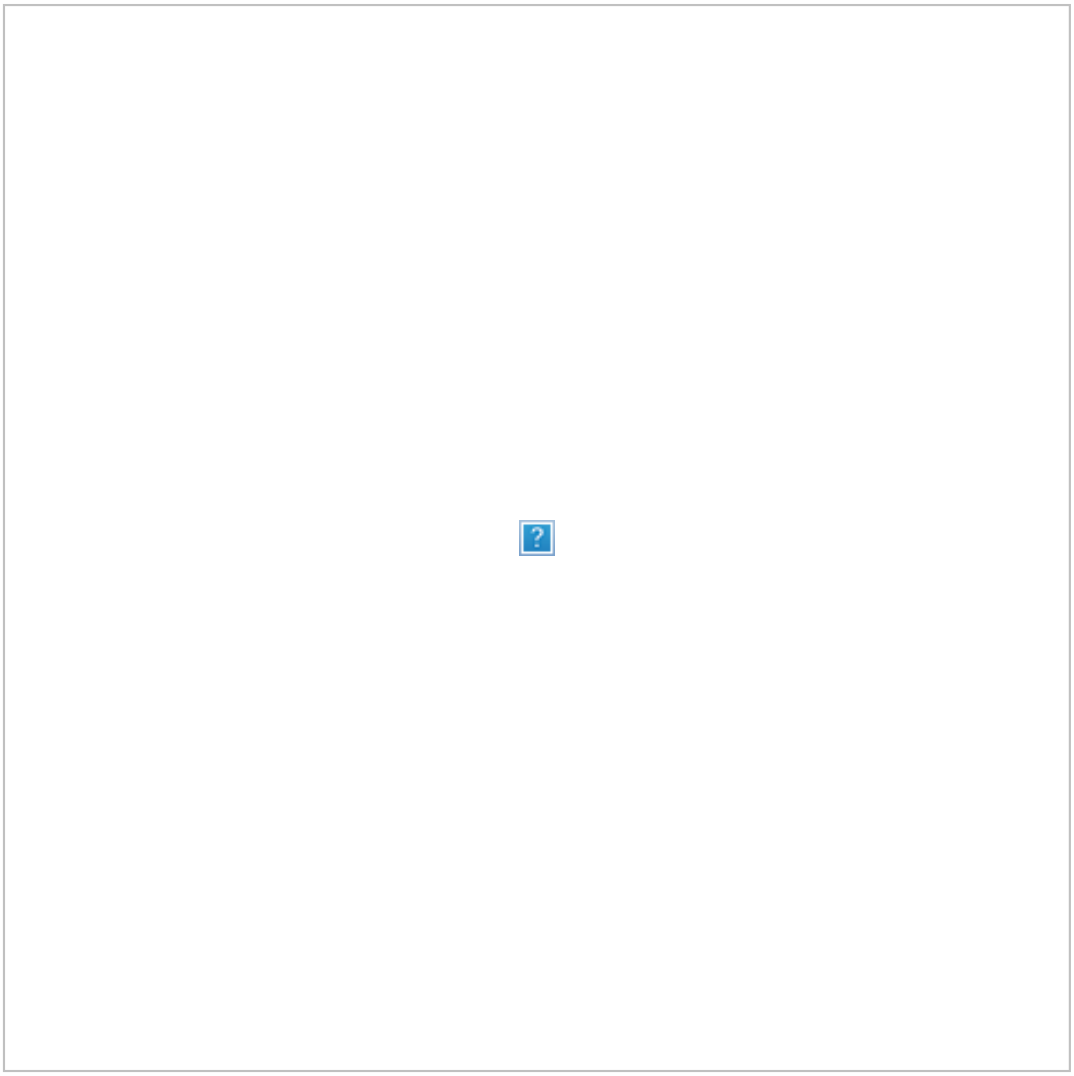
属性与方法

每个 Vue 实例都会**代理**其 `data` 对象里所有的属性

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Document</title>
    <script type="text/javascript" src="js/vue.js"></script>
</head>
<body>
    <div id="app">
        <!-- 允许采用简洁的模板语法来声明式的将数据渲染进 DOM -->
        {{message}}
    </div>
</body>
<script type="text/javascript">
    var app = new Vue({
        el:"#app",
        data:{
            message:'hello Vue!'
        }
    });
  //代理
    console.log(app.message);
</script>
</html>
```

实例的生命周期

每个 Vue 实例在被创建之前都要经过一系列的初始化过程。例如，实例需要配置数据观测(data observer)、编译模版、挂载实例到 DOM ， 然后在数据变化时更新 DOM 。在这个过程中，实例也会调用一些 **生命周期钩子** ， 这就给我们提供了执行自定义逻辑的机会，例如， `created` 这个钩子在实例被创建之后被调用



生命周期函数：

- beforeCreate 在实例初始化之后，数据观测(data observer) 和 event/watcher 事件配置之前被调用
- created 实例已经创建完成之后被调用。在这一步，实例已完成以下的配置：数据观测(data observer)，属性和方法的运算， watch/event 事件回调。然而，挂载阶段还没开始，`$el` 属性目前不可见
- beforeMount 在挂载开始之前被调用：相关的 `render` 函数首次被调用
- mounted `el` 被新创建的 `vm.$el` 替换，并挂载到实例上去之后调用该钩子。如果 root 实例挂载了一个文档内元素，当 `mounted` 被调用时 `vm.$el` 也在文档内
- beforeUpdate 数据更新时调用，发生在虚拟 DOM 重新渲染和打补丁之前。
- updated 由于数据更改导致的虚拟 DOM 重新渲染和打补丁，在这之后会调用该钩子。
- activated keep-alive 组件激活时调用
- deactivated keep-alive 组件停用时调用
- beforeDestroy 实例销毁之前调用。在这一步，实例仍然完全可用
- destroyed Vue 实例销毁后调用。调用后，Vue 实例指示的所有东西都会解绑定，所有的事件监听器会被移除，所有的子实例也会被销毁。

计算属性和数据监听

计算属性

模板内的表达式是非常便利的，但是它们实际上只用于简单的运算。在模板中放入太多的逻辑会让模板过重且难以维护.

它也是一个vue实例的属性，但是它可以执行复杂的逻辑处理。并且在计算属性内如果依赖了某个data数据，当数据发生改变的时候计算属性也会更新，实例：

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Document</title>
    <script type="text/javascript" src="js/vue.js"></script>
</head>
<body>
    <div id="app">
        <p>原始的message:{{message}}</p>
        <p>翻转后的message:{{reversedMessage}}</p>
    </div>

</body>
<script type="text/javascript">

    var app = new Vue({
        el:"#app",
        data:{
            message:"Hello Vue"
        },
        computed:{
            reversedMessage:function (){
                /*
                    计算属性:reversedMessage是用作vm.reversedMessage的getter.
                    getter:用于获得属性值的方法
                */
                // 这个每次返回的都是同一个值，因为该函数中并没有依赖data中的任何内容
                // return Math.random();
                return this.message.split("").reverse().join("");
            }
        }
    });
</script>
</html>
```

当然如果并不依赖某个数据，那么计算属性也不会变化，比如计算属性中是获取当前时间`Date.now()`就只是获取第一次获取的时间

对比methods

我们可以将同一函数定义为一个 method 而不是一个计算属性。对于最终的结果，两种方式确实是相同的。然而，不同的是**计算属性是基于它们的依赖进行缓存的**。计算属性只有在它的相关依赖发生改变时才会重新求值。这就意味着只要 `message` 还没有发生改变，多次访问 `now2` 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。而method 调用**总会**执行该函数,可以在控制台中访问 `vm.now()` 返回的最新时间，而 `vm.now2` 计算属性返回的都是固定的时间，只有在 `vm.now2` 所依赖的 `message` 发生改变的时候 `vm.now2` 才会更新.

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Document</title>
    <style type="text/css">
        .a-enter-active, .a-leave-active {
            transition: opacity .5s
        }
        .a-enter, .a-leave-active {
            opacity: 0
        }
    </style>
    <script type="text/javascript" src="vue.js"></script>
</head>
<body>
    <div id="app">
        <div>时间: {{now()}}</div>
    </div>
</body>
<script type="text/javascript">

    var vm = new Vue({
        el:"#app",
        data:{
           message:"hello vue!"
        },
        /*
        每次调用都执行内部的代码
        */
        methods:{
            now:function(){
                return this.message+","+Date.now()
            }
        },
        /*
        计算属性只有在它的相关依赖发生改变时才会重新求值
        */
        computed:{
            now2:function() {
```

```
                var a = this.message;
                return Date.now();
            }
        }
    });
</script>
</html>
```

数据监听

除了可以用计算属性去监听数据的变化以外，`vue`还提供了`watch`这个更加通用的选项

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Document</title>
    <script type="text/javascript" src="js/vue.js"></script>
</head>
<body>
    <div id="app">
        <div>{{fullName}}</div>
    </div>

</body>
<script type="text/javascript">

    var app = new Vue({
        el:"#app",
        data:{
            firstName:'唐',
            lastName:'菜也',
            fullName:'唐菜也'
        },
        watch:{
            //监听firstName的变化,都接受两个参数，第一个为新值，第二个为老值
            firstName:function (val){
                // console.log(val);
                this.fullName = val+this.lastName;
            },
            lastName:function (val){
                this.fullName = this.firstName+val;
            }
        }
    });


</script>
</html>
```

如果是使用计算属性来实现：

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Document</title>
    <script type="text/javascript" src="js/vue.js"></script>
</head>
<body>
    <div id="app">
        <div>{{fullName}}</div>
    </div>

</body>
<script type="text/javascript">

    var app = new Vue({
        el:"#app",
        data:{
            firstName:'唐',
            lastName:'菜也'
        },
        /*watch:{
            //监听firstName的变化
            firstName:function (val){
                // console.log(val);
                this.fullName = val+this.lastName;
            },
            lastName:function (val){
                this.fullName = this.firstName+val;
            }
        }*/
        computed:{
            fullName:function (){
                return this.firstName+this.lastName;
            }
        }
    });


</script>
</html>
```

是不是更好些?

watcher提供的是更加通用的方法，只是监听的属性发生变化都会调用对应的方法，比如在属性发生改变的时候要去调用某个方法.

事件处理器

监听事件

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

```
<div id="example-1">
  <button v-on:click="counter += 1">增加 1</button>
  <p>这个按钮被点击了 {{ counter }} 次。</p>
</div>

var example1 = new Vue({
  el: '#example-1',
  data: {
    counter: 0
  }
})
```

方法处理器

`v-on` 可以接收一个定义的方法来调用。

```
<div id="example-2">
  <!-- `greet` 是在下面定义的方法名 -->
  <button v-on:click="greet">Greet</button>
</div>
var example2 = new Vue({
  el: '#example-2',
  data: {
    name: 'Vue.js'
  },
  // 在 `methods` 对象中定义方法
  methods: {
    greet: function (event) {
      // `this` 在方法里指当前 Vue 实例
      alert('Hello ' + this.name + '!')
      // `event` 是原生 DOM 事件
      alert(event.target.tagName)
    }
  }
})
// 也可以用 JavaScript 直接调用方法
example2.greet() // -> 'Hello Vue.js!'
```

内联方法

```
<div id="example-3">
  <button v-on:click="say('hi')">Say hi</button>
  <button v-on:click="say('what')">Say what</button>
</div>
new Vue({
  el: '#example-3',
  methods: {
    say: function (message) {
      alert(message)
    }
  }
})
```

事件修饰符

在事件处理程序中调用 `event.preventDefault()` 或 `event.stopPropagation()` 是非常常见的需求。尽管我们可以在 methods 中轻松实现这点，但更好的方式是：methods 只有纯粹的数据逻辑，而不是去处理 DOM 事件细节。

为了解决这个问题， Vue.js 为 `v-on` 提供了 **事件修饰符**。通过由点(.)表示的指令后缀来调用修饰符。

- `.stop`
- `.prevent`
- `.capture`
- `.self`
- `.once`

```
<!-- 阻止单击事件冒泡 -->
<a v-on:click.stop="doThis"></a>

<!-- 提交事件不再重载页面 -->
<form v-on:submit.prevent="onSubmit"></form>

<!-- 修饰符可以串联  -->
<a v-on:click.stop.prevent="doThat"></a>

<!-- 只有修饰符 -->
<form v-on:submit.prevent></form>

<!-- 添加事件侦听器时使用事件捕获模式 -->
<div v-on:click.capture="doThis">...</div>

<!-- 只当事件在该元素本身（而不是子元素）触发时触发回调 -->
<div v-on:click.self="doThat">...</div>
```

按键修饰符

在监听键盘事件时，我们经常需要监测常见的键值。 Vue 允许为 `v-on` 在监听键盘事件时添加按键修饰符：

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

记住所有的 keyCode 比较困难，所以 Vue 为最常用的按键提供了别名：

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

<!-- 缩写语法 -->
<input @keyup.enter="submit">
```

全部的按键别名：

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

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

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