1. 基础
   1. 模板语法
      1. 定义模板方式

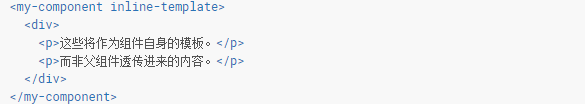
（1）在组件template属性中定义



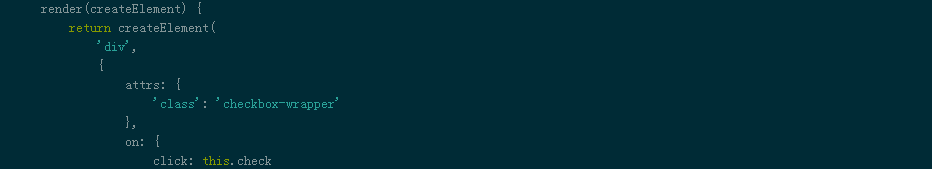
（2）定义到挂载html文件中的script下，该script标签使用text/x-template标记，并由组件定义的id引用。



（3）内联模板，即，如果子组件有inline-template特性，组件将把它的内容当作它的模板，而不是把它当作分发内容。这让模板编写起来更灵活。



（4）在render函数中定义。



render函数中还可以使用jsx语法。



（5）最常用最方便的还是单文件组件，定义在template中。



* + 1. 插值语法

**1.普通插值**

（1）使用双大括号（Mustache语法）插入文本。可以使用v-once指令执行一次性插值（即，绑定的数据即使发生改变，插值处的内容也不会更新）。



（2）{{}}中可以使用JavaScript表达式，但不能使用语句。

**2.特性插值**

（1）对于标签特性，使用v-bind指令插值，不需要使用Mustache语法（使用的话岂不多此一举）。



（2）可以一次绑定多个特性。



（3）如果变量值为null、undefined、false，特性甚至不会显示。

**3.class**

对于class、style，可以像上面那样使用特性插值，但还有一些特殊使用方法。

（1）使用对象形式，用于动态切换class（注意是切换用）。



如上，active是1个class。数据属性isActive的值为true，div将会有active这个class。

（2）可以使用1个数据对象。



（3）可以和普通class混用。



（4）可以使用数组，表示多个class。





（5）在自定义组件上使用class属性时，这些class将被添加到该组件的根元素上面，不会被覆盖。







**4.style**

与class类似，style还有如下语法：

（1）可以使用对象形式设置多个内联样式。





（2）也可以使用数组。



（3）Vue.js会自动侦测并添加相应的浏览器引擎前缀。

**5.插入原始HTML代码**

（1）使用v-html指令插入原始HTML代码。





注意，动态渲染HTML可能非常危险，很容易导致XSS攻击，绝不要对用户提供的内容使用插值。

（2）不能使用v-html来复合局部模板。组件更适合作为可重用和可组合的基本单位。

* + 1. 指令

**1.概述**

（1）带有v-前缀的特殊属性就是指令。

（2）修饰符为.的特殊后缀，用于指示一个指令应以特殊方式绑定。



（3）Vue.js为v-bind和v-on这两个最常用的指令提供了简写:和@。

**2.条件渲染**

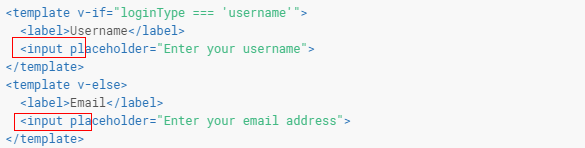
（1）使用v-if、v-else-if、v-else指令，需要配对。或使用v-show指令。v-show为根据表达式值的真假，切换元素的display属性，不支持<template>元素，也不支持v-else指令。



（2）指令必须添加到一个元素上，对于多个元素，可以使用<template>作为不可见的包裹元素。



（3）由于vue会复用元素，如下，当用户切换2个input时，用户输入也会跟着切换。如果不想这样，可以添加一个唯一值的key属性。



**3.列表渲染**

（1）v-for指令根据一组选项列表进行渲染（放在循环的标签上）。



（2）在v-for块中，有对父作用域属性的完全访问权限。v-for还支持可选的第二个参数，作为当前项的索引。



（3）可以使用of代替in，以便使用迭代器。

（4）也可以作用于对象，按Object.keys()的结果遍历。



（5）也可以在<template>上使用v-for渲染多个元素。

（6）如果处于同一节点，v-for的优先级比v-if更高，即，v-if将分别重复运行于每个v-for循环中。



如果目的是有条件地跳过循环的执行，可以将v-if置于外层元素或[<template>](https://cn.vuejs.org/v2/guide/conditional.html#%E5%9C%A8-lt-template-gt-%E4%B8%AD%E9%85%8D%E5%90%88-v-if-%E6%9D%A1%E4%BB%B6%E6%B8%B2%E6%9F%93%E4%B8%80%E6%95%B4%E7%BB%84)上。

（7）在组件中使用v-for时，key现在是必须的，且需要使用props，因为组件有自己独立的作用域。



（8）对于已渲染过的元素，vue默认使用“就地复用”策略。即，数据项顺序改变，vue不会移动dom来匹配顺序，而是简单复用此处每个元素，确保在特定索引下，显示已被渲染过的每个元素。即，如果第2个checkbox被选中，即使调整顺序或增加元素，还是第2个checkbox被选中。

可以添加key来改变这种行为。

**4.表单输入绑定**

（1）可以使用v-model指令在<input>及<textarea>元素上创建双向数据绑定。

（2）v-model会忽略元素的value、checked、selected特性的初始值，而总是将Vue实例的数据作为数据来源，因此应该通过JavaScript在组件的data选项中声明初始值。

（3）对于如中文、日文等需要使用输入法的语言，v-model不会起作用，此时需要使用input事件。

* + 1. DOM事件

（1）使用v-on指令监听DOM事件。

（2）事件句柄函数可以只写函数名，会自动将事件对象传入。





（3）事件句柄还可以写作内联调用方法形式，如果需要传入事件对象，需要手动传入$event。



（4）既然可以写入内联调用方法，因此也可以写入语句。如下，hovering就是data中的变量。



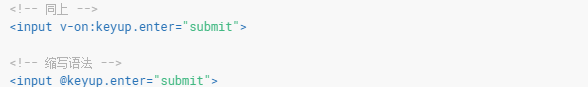
（5）Vue.js为v-on提供.stop、.prevent、.capture、.self、.once等事件修饰符，可实现如同event.preventDefault或event.stopPropagation等效果。



（6）可以使用按键修饰符监听键盘事件。



Vue还为最常用的按键提供了别名。



* 1. 组件
     1. 概要

https://segmentfault.com/a/1190000010095089

http://blog.csdn.net/gao531162436/article/details/78732114

https://segmentfault.com/a/1190000010527064

https://www.cnblogs.com/diantao/p/5848618.html

https://www.w3cplus.com/blog/tags/632.html

https://www.w3cplus.com/blog/vue?page=1

（1）组件可用来扩展HTML元素，封装可重用的代码。所有的Vue组件同时也都是Vue的实例。

（2）注册一个全局组件。需要确保在初始化根实例之前注册组件。



（3）可以在Vue实例的components选项中注册，则为其作用域下的组件。



（4）注意dom元素限制，如下，自定义组件<my-row>会被当作无效的内容，变通的方案是使用特殊的is特性。





（5）组件注册中的data必须是函数，如下将报错。



如果函数返回1个引用类型，如果有多个组件实例，将导致它们共用这个引用类型值而出现混淆。



应该每个组件返回1个对象。



* + 1. 数据

（1）data会在组件实例化的时候被劫持为观察者对象。模板中使用的数据会被注册为订阅者。

（2）data中的数据就是一些简单的数据，变化的时候就更新其他依赖它的数据（不见得会更新dom，得要引起dom变化才会）。

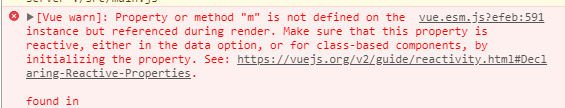
（3）尽管可以在模板内使用表达式，但在模板中放入太多逻辑会让模板过重且难以维护。因此出现了计算属性。

计算属性本质也是1个观察者，默认会转为1个getter，监听的对象是对应的data。计算属性可以设置setter，注意，没法直接更改计算属性的值，因为它不是1个data，只能更新计算属性依赖的data。如下，props也不能直接更新，需要在父组件内更新，这里就使用组件通信更新父组件中的data。

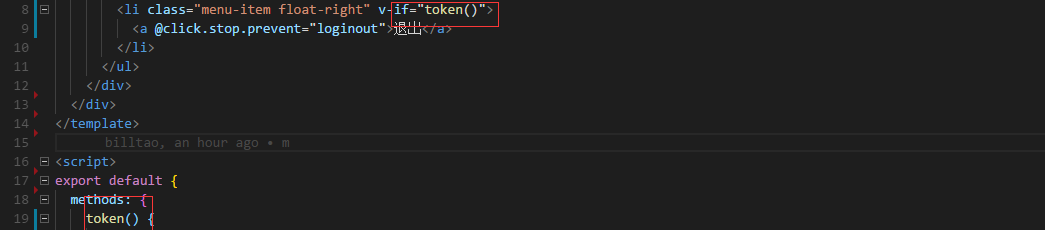


（4）data、computed都是观察者。他们变化了，就会引起保存的订阅者变化。这里的订阅者也就是模板编译得到的那些。如果想设置额外的订阅者，就需要使用watch选项。注意，watch增加的不是1个观察者，而是1个订阅者。如下，如果没有变量m，将报错。





（5）除此之外，还可以使用调用方法来达到计算属性同样效果，注意，使用时要加括号。



计算属性和计算方法的区别：计算属性会基于依赖缓存，只有依赖发生改变时才会重新求值。而计算方法在每次重新渲染触发时，总是执行。

* + 1. 组件通信

prop向下传递，事件向上传递。

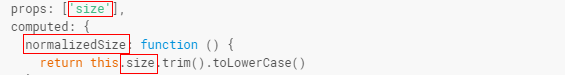
**1.使用prop传递数据**

（1）组件实例的作用域是孤立的。父组件的数据需要通过prop才能下发到子组件中。子组件要显式地用props选项声明它预期的数据。





（2）数据流是单向绑定。不应在子组件中修改传入数据，而应该使用计算属性。



（3）props中可以使用驼峰式对应模板中使用-连接传入的变量。





（4）可以用v-bind动态绑定prop到父组件的数据。每当父组件的数据变化，该变化也会传导给子组件。



也可以绑定1个对象。

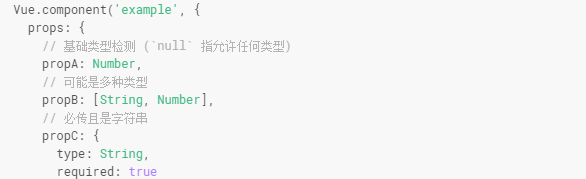


（5）如果传递指定数据类型的数据，不应使用字面量语法，而应该使用绑定。





（6）可以为组件的prop指定验证规则。



（8）在组件上添加的非props属性，会被直接附加到组件的根元素上。



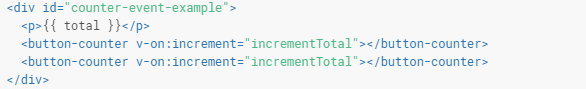
**3.自定义事件**

（1）vue中的事件分为两类：DOM事件和自定义事件。在组件上绑定的事件就是自定义事件，在组件上绑定DOM事件默认无效，如下，click事件无效，即使组件内元素没有阻止冒泡，除非加.native修饰符。





（2）每个Vue实例都实现了事件接口，即：使用$on(eventName)监听事件，使用$emit(eventName)触发事件。不能用$on侦听子组件释放的事件，而必须在模板里直接用v-on绑定。







（3）2.3起重新引入了.sync修饰符，作为一个编译时的语法糖，用于自动更新父组件属性的v-on监听器。如：



会被扩展为：



当子组件更新foo值时，需要显式触发一个更新事件：



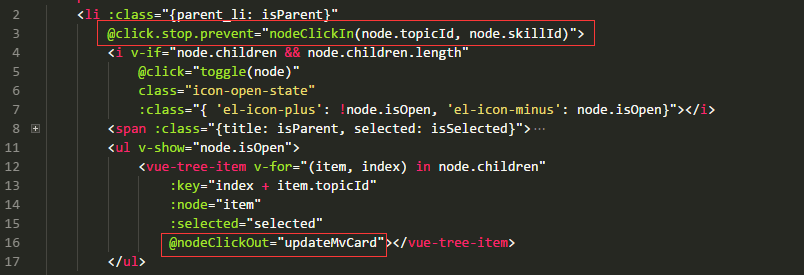
（4）可以使用自定义事件创建自定义的表单输入组件，可使用v-model实现数据双向绑定。

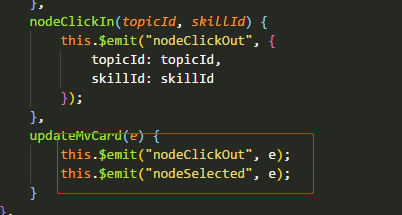


v-model不过是v-bind:value和v-on:input的语法糖。



（5）递归组件事件触发。如下，因为是递归，不知道有多少层，因此，外层的上层仍然可能还是递归组件，因此事件回调内还得继续触发组件的事件。





**4.非父子关系组件之间通信**

可以使用一个空的Vue实例作为事件总线。

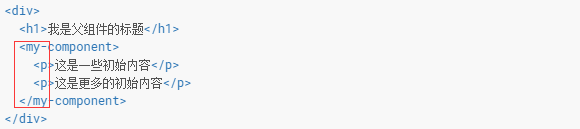
**5.使用插槽分发内容**

（1）在父组件中，将内容传入子组件XML中，并在子组件中显示的过程被称为内容分发。

（2）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。除非子组件模板包含有<slot>插口，否则父组件的内容将会被丢弃。

（3）当子组件模板只有一个没有属性的插槽时，父组件传入的整个内容片段将插入到插槽所在的DOM位置，并替换掉插槽标签本身。最初在<slot>标签中的任何内容都被视为备用内容。





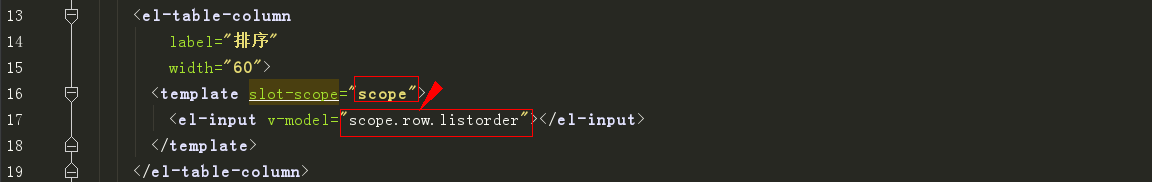


（4）<slot>元素可以使用name进一步配置如何分发内容，即具名插槽。

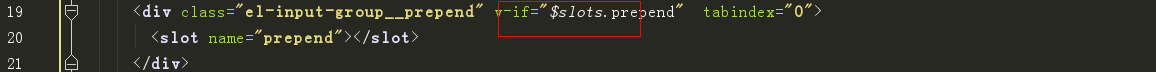




（5）作用域插槽是一种特殊类型的插槽。父级组件中需要1个有slot-scope特殊特性的<template>元素，表示为作用域插槽的模板。slot-scope的值将被用作一个临时变量名，用于接收从子组件传递过来的prop对象。然后，在template中的元素可以使用这个变量。



（6）组件实例中，可以使用$slots访问所有具名插槽。





**6.动态组件**

（1）通过使用保留的<component>元素，动态地绑定其is特性，可以让多个组件使用同一个挂载点，并动态切换。

（2）可以添加一个keep-alive指令参数，将切换出去的组件保留在内存中，以保留它的状态，或避免重新渲染。

**7.其他**

（1）可以使用ref为子组件指定一个引用ID，以便在JavaScript中直接访问子组件。

（2）Vue.js允许将组件定义为一个工厂函数，异步解析组件的定义。Vue.js只在组件需要渲染时触发工厂函数，并且把结果缓存起来，用于后面的再次渲染。



（3）组件命名约定：在HTML模板中，使用kebab-case（短横线分隔命名）。

（4）如果组件未经slot元素传入内容，可以在组件名后使用/使其自闭合。

（5）组件可以在它的模板内递归地调用自己。只有当它有name选项时才可以这么做。

（6）可以使用text/x-template类型的JavaScript标签，并且指定一个id定义模板。





* + 1. 混合

（1）混合可用于复用组件功能。当组件使用混合对象时，所有混合对象的选项将被混入该组件本身的选项。

（2）当组件和混合对象含有同名选项时，这些选项会以恰当的方式混合。如，同名钩子函数将混合为一个数组，因此都将被调用。值为对象的选项，如methods、components、directives，将被混合为同一个对象，冲突的键名取组件对象的键值对。

（3）使用全局混合对象，将会影响到所有之后创建的Vue实例。



* + 1. 单文件组件

（1）中小规模的项目中，可以使用Vue.component定义全局组件，然后使用new Vue({el:'#container'})在每个页面内指定一个容器元素。

（2）在更复杂的项目中，可以使用为.vue扩展名的单文件组件。

1. 路由
   * 1. 路由原理

**1.history模式**

（1）h5提供pushState和replaceState更改浏览器地址栏，且不会触发浏览器reload。注意，两个api只是修改地址栏，页面并没有变，要想更改页面，还得使用dom操作。



（2）因pushState同时将url压入历史记录栈，因此可以通过监听popstate事件，来监听用户点击浏览器后退、前进操作，此时也不会发生浏览器reload。



（3）因浏览器提供go、back方法，因此还可以模拟用户点击浏览器的前进后退按钮。

（4）如何实现路由：通过给a标签等绑定1个事件，阻止其默认行为，然后使用pushState和replaceState更改浏览器地址栏，并执行dom操作等。

2.**hash模式**

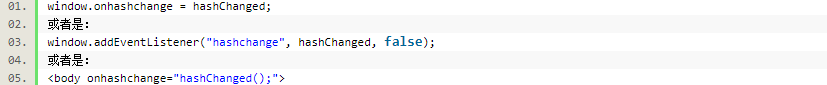
（1）#代表网页中的一个位置。其右面的字符，就是该位置的标识符。#是用来指导浏览器动作的，对服务器端完全无用。所以，HTTP请求中不包括#。如访问如下地址，发生给浏览器的请问为：





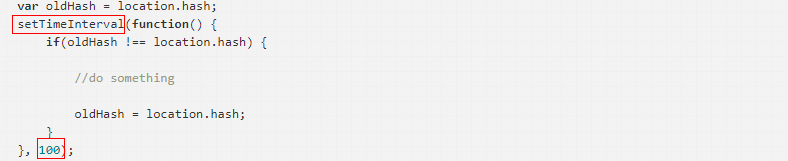
（2）单单改变#后的部分，浏览器只会滚动到相应位置，不会重新加载网页，但都会在浏览器的访问历史中增加一个记录。使用"后退"按钮，就可以回到上一个位置。不过对应IE6和IE7，不会增加历史记录。

（3）IE8+支持h5的onhashchange事件。

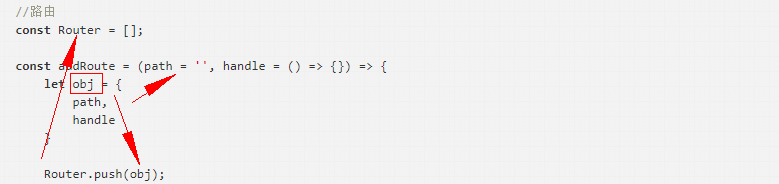


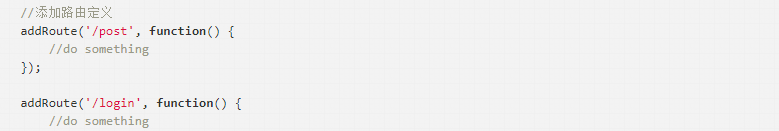
（4）因此，同样可以阻止a的默认行为，通过js修改hash，操作dom，实现页面更改。

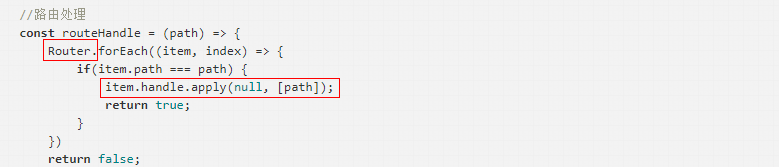
（5）对于IE6和IE7，则可以使用轮询来监听hash变化，进而使用js操作页面。



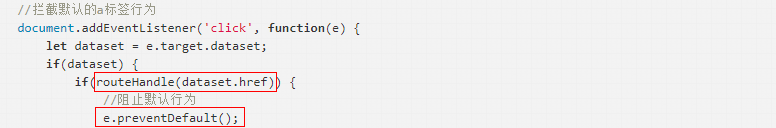
**3.路由实现**











**4.vue中的hash和history模式**

（1）为构建SPA（单页面应用），需要引入前端路由系统，这也就是Vue-Router存在的意义。前端路由的核心在于改变视图的同时不会向后端发出请求。

（2）浏览器当前提供了2种支持：

1）hash模式，格式如下：



2）history模式，利用了HTML5 HistoryInterface中新增的pushState()和replaceState()方法。只是当它们执行修改时，虽然改变了当前的URL，但浏览器不会立即向后端发送请求。格式如下：



（3）一般场景下，hash和history都可以，除非你更在意颜值，#符号夹杂在URL里看起来确实有些不太美丽。

（4）调用history.pushState相比于直接修改hash，存在以下优势：1）pushState设置的新URL可以是与当前URL同源的任意URL，而hash只可修改#后面的部分，因此只能设置与当前URL同文档的URL。2）pushState设置的新URL可以与当前URL一模一样，这样也会把记录添加到栈中，而hash设置的新值必须与原来不一样才会触发动作将记录添加到栈中。3）pushState通过stateObject参数可以添加任意类型的数据到记录中，而hash只可添加短字符串。4）pushState可额外设置title属性供后续使用。

（5）hash模式下，仅hash符号之前的内容会被包含在请求中，如http://www.abc.com。因此对于后端来说，即使没有做到对路由的全覆盖，也不会返回404错误。

（6）history模式下，前端的URL必须和实际向后端发起请求的URL一致，如http://www.abc.com/book/id。如果后端缺少对/book/id的路由处理，将返回404错误。Vue-Router官网里如此描述：“不过这种模式要玩好，还需要后台配置支持……所以呢，你要在服务端增加一个覆盖所有情况的候选资源：如果URL匹配不到任何静态资源，则应该返回同一个index.html页面，这个页面就是你app依赖的页面。”

* + 1. 基础

**1.概要**

（1）首先需要使用Vue.use(VueRouter)载入路由，然后创建router实例，定义路由和组件的映射关系，然后挂载根实例中。



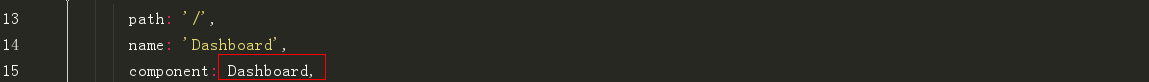




（2）使用router-link组件来导航，<router-link>会被渲染成一个<a>标签。匹配到的组件将渲染在<router-view>所在位置。

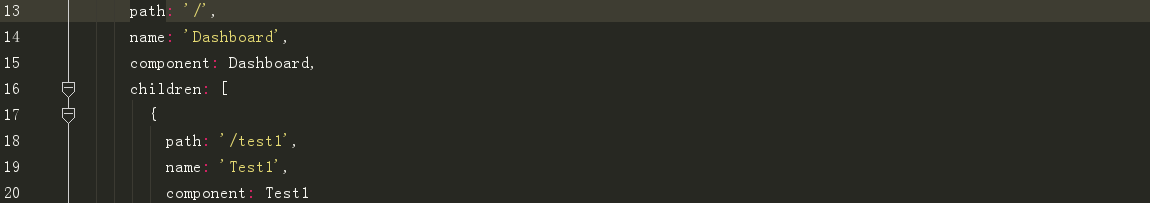
如下，跟路由匹配到的Dashboard组件将会被渲染到<router-view>所在位置。





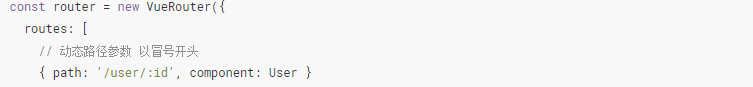
注意，<router-view>可以有多个也不会报错，只是会重复渲染。

（3）Dashboard组件中还可以继续有<router-view/>，如果把路由规则放在跟路径路由的children下，则对应路径匹配到的组件将不会被渲染在跟路径下的<router-view/>，而是在Dashboard组件中的<router-view/>，即嵌套路由。



**2.动态路由**

（1）动态路径参数以:开头。



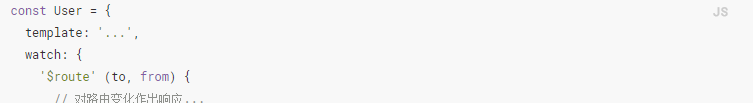
（2）当匹配到一个路由时，参数值会被设置到this.$route.params，可以在每个组件内使用。



可以在一个路由中设置多段路径参数，对应的值都会设置到$route.params中。



（3）当使用路由参数时，如从/user/foo导航到user/bar，原来的组件实例会被复用。这也意味着组件的生命周期钩子不会再被调用。如果想对路由参数的变化作出响应，可使用watch监测$route对象。



**4.命名路由**

（1）使用名称标识路由，在链接一个路由，或执行跳转时有用。



**5.命名视图**

1个页面内展示多个视图，就得有多个router-view，如何将对应的组件渲染到对应的router-view中，就需要命名视图。

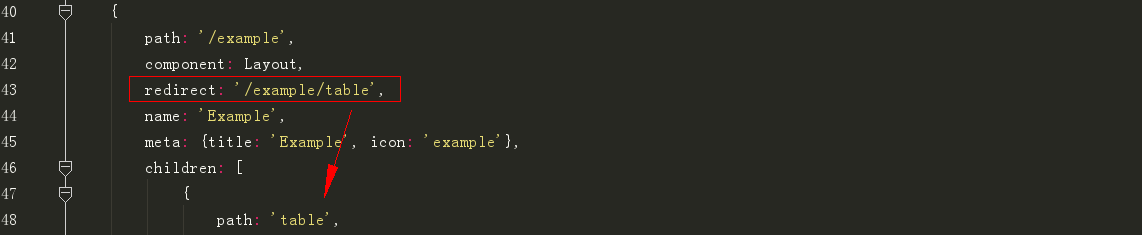




在组件中使用$route，会使组件和对应的路由形成高度耦合，可以使用props将组件和路由解耦。

**6.重定向**

即跳转到1个路由的时候，会重定向到另外1个路由。



* + 1. API

**1.构造函数配置**



（1）base：设置基路径。

（2）linkActiveClass：全局配置<router-link>的默认激活class类名。

（3）parseQuery/stringifyQuery：提供自定义查询字符串的解析/反解析函数。

（4）fallback：当浏览器不支持history.pushState控制路由是否应该回退到hash模式。默认值为true。在IE9中，设置为false会使得每个router-link导航都触发整页刷新。它可用于工作在IE9下的服务端渲染应用，因为一个hash模式的URL并不支持服务端渲染。

**2.实例属性/方法**

（1）currentRoute。

（2）beforeEach/beforeResolve。

（3）push/replace/go/back/forward/resolve。

（4）addRoutes。

**3.路由信息对象**

（1）表示当前激活的路由的状态信息，包含了当前URL解析得到的信息，还有URL匹配到的routerecords（路由记录）。

（2）routeobject是immutable（不可变）的，每次成功的导航后都会产生一个新的对象。

（3）path、params、query等值。

（4）可以在在组件内，通过this.$route获取等。

**4.<router-link>**

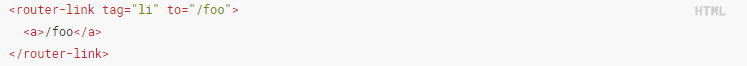
<router-link>比起写死的<a>会好一些，理由如下：

（1）无论是HTML5history模式还是hash模式，它的表现行为一致，所以，当你要切换路由模式，或者在IE9降级使用hash模式，无须作任何变动。

（2）在HTML5history模式下，router-link会守卫点击事件，让浏览器不再重新加载页面（也就是写a标签需要自己阻止其默认行为）。

（3）当你在HTML5history模式下使用base选项之后，所有的to属性都不需要写（基路径）了。

还可以将激活class应用在外层元素（li）。



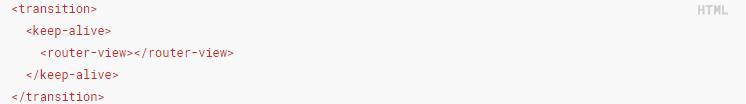
（4）路由to也可以绑定变量，需要使用v-bind:to或者简写:to。



**5.<router-view>**

（1）<router-view>用于渲染路径匹配到的视图组件。<router-view>渲染的组件还可以内嵌自己的 <router-view>，根据嵌套路径，渲染嵌套组件。

（2）它也是个组件，所以可以配合<transition>和<keep-alive>使用。



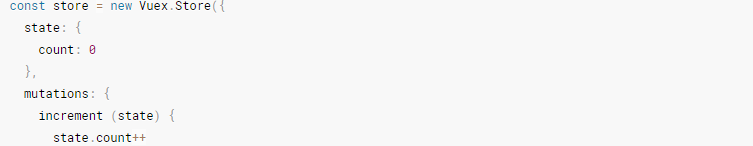
1. vuex

**1.概要**

（1）Vuex采用集中式存储管理应用的所有组件的状态，并保证状态以一种可预测的方式发生变化。

（2）Vuex应用的核心就是store。store是一个容器，包含应用中大部分的state。

（3）Vuex和单纯的全局对象有两点不同：1）Vuex的状态存储是响应式的。当Vue组件从store中读取到的状态发生变化，相应的组件也会更新。2）不能直接改变store中的状态。改变状态唯一途径是显式提交mutation，目的是可以方便地跟踪每一个状态的变化。





**2.state**

（1）Vuex使用单一状态树，即一个对象就包含了全部的应用层级状态。从store实例中读取状态最简单的方法就是在计算属性中返回某个状态（先引入）。如下，每当store.state.count变化的时候,都会重新求取计算属性，并且触发更新相关联的DOM。



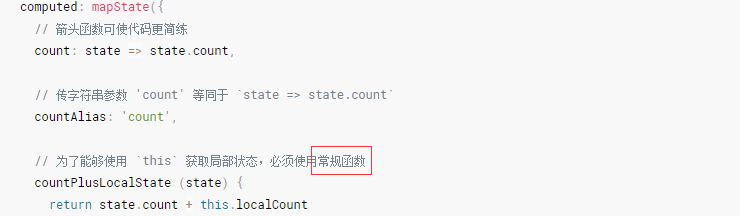
（2）如果组件多，上面的方法显得非常麻烦。可以直接在根组件中将store注入到每一个子组件中。



子组件可以通过this.$store访问到状态。



（3）当组件需要获取多个状态时，都声明为计算属性显得重复和冗余。可以使用mapState辅助函数生成计算属性。



当映射的计算属性与state子节点同名时，可以如下声明。



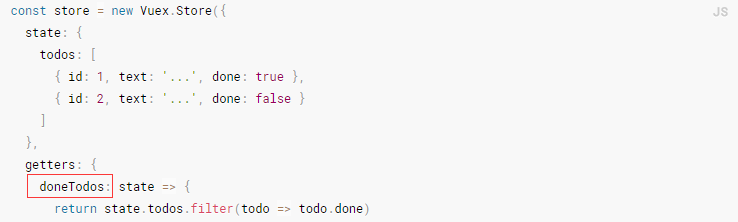
（4）mapState函数返回一个对象，可以使用对象展开运算符，使其与局部计算属性混合。



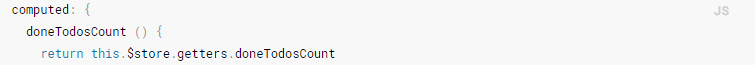
（5）使用Vuex并不意味着需要将所有的状态放入Vuex。如果有些状态严格属于单个组件，最好还是作为组件的局部状态。

**3.Getter**

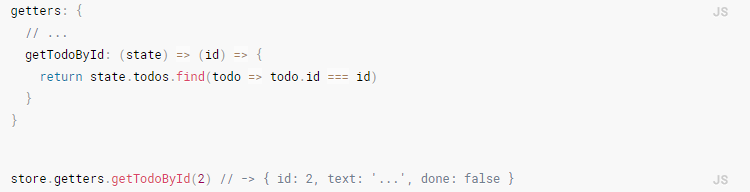
（1）当多个组件需要从store中的state中派生出一些状态，可以在store中定义getter（可以认为是store的计算属性）。



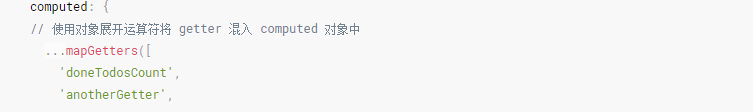
（2）在任何组件中可以获取getter。



（3）可以通过让getter返回一个函数，来实现给getter传参。



（4）mapGetters辅助函数可将getter映射到局部计算属性。

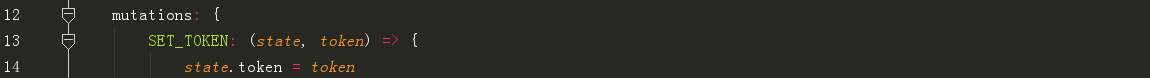


使用对象形式对getter属性起1个别名。



**4.Mutation**

（1）更改store中的状态的唯一方法是提交mutation。mutation类似于事件，有一个字符串的事件类型和一个回调函数。回调函数就是实际进行状态更改的地方，它接受state作为第一个参数。



（2）使用store.commit方法触发type对应的handler。



（3）可以向store.commit传入额外的参数，即mutation的载荷（payload）。



（4）可以在组件中使用this.$store.commit('xxx')提交mutation，或使用mapMutations辅助函数将组件中的methods映射为store.commit调用。

（5）最好提前在store中初始化好所有所需属性。当添加新属性时，应使用Vue.set(obj,'newProp',123),或者以新对象替换老对象。



（6）mutation都是同步事务。

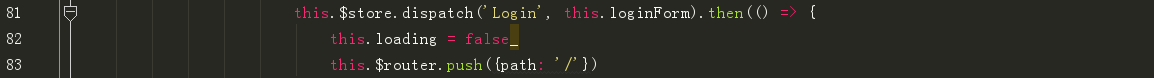
**5.Action**

（1）Action类似于mutation，不同在于：Action提交的是mutation，而不是直接变更状态。Action可以包含任意异步操作。

（2）context对象与store实例具有相同方法和属性的，因此可以调用context.commit提交一个mutation，或者通过context.state和context.getters来获取state和getters。







（3）可以用参数解构来简化代码



（4）Action通过store.dispatch方法触发。



目的是可以在action中定义异步操作。



（5）在组件中可以使用mapActions辅助函数将组件的methods映射为store.dispatch调用。



**6.Module**

（1）Vuex允许我们将store分割成模块（module）。每个模块拥有自己的state、mutation、action、getter、甚至是嵌套子模块。

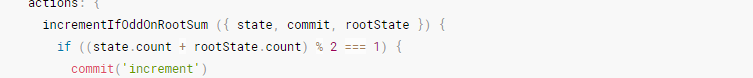




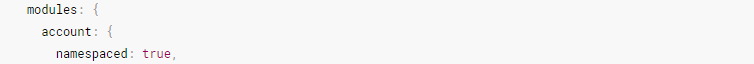


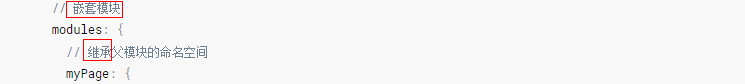
（2）对于模块内部的mutation和getter，接收的第一个参数是模块的局部状态对象，根节点状态会作为第3个参数暴露出来。同样，对于模块内部的action，局部状态通过context.state暴露出来，根节点状态则为context.rootState。



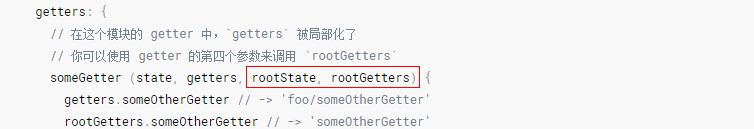


（3）默认情况下，模块内部的action、mutation和getter是注册在全局命名空间的——这样使得多个模块能够对同一mutation或action作出响应。如果希望你的模块具有更高的封装度和复用性，你可以通过添加namespaced:true的方式使其成为命名空间模块。





（4）在命名空间模块内访问全局内容（GlobalAssets）中，如果你希望使用全局state和getter，rootState和rootGetter会作为第三和第四参数传入getter，也会通过context对象的属性传入action。



（5）在store创建之后，你可以使用store.registerModule方法动态注册模块。



1. 专题
   * 1. 原理

（1）一个Vue应用由一个通过new Vue创建的根Vue实例，以及可选的嵌套的、可复用的组件树组成。

（2）Vue的data对象包含所有的属性。当属性值发生改变时，视图将会产生“响应”，即匹配更新为新的值。但是，只有在实例被创建时data中就存在的属性才是响应式的，对于新添加的属性，改动将不会触发任何视图更新。可以使用占位。

（3）Vue实例暴露了一些有用的实例属性与方法，有前缀$，以便与用户定义的属性区分开来。

**1.Vue构造器及其实例化**

（1）Vue.js是一个构造函数，在编程中称为构造器。每一个new Vue()都是一个Vue构造函数的实例，即实例化过程。Vue构造器要求实例化时需要传入一个选项对象，选项对象包括数据(data)，模板(tamplate)，挂载元素(el)，方法(methods)，生命周期钩子等选项。

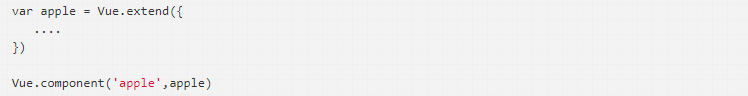
（2）data会代理其对象里的所有属性。只有data里的属性是响应式的，即通信是双向的。

（3）Vue实例暴露了一些有用的实例属性与方法，为与代理属性（data中的）相区分，前面添加了前缀$，如：app.$data===data//->true；app.$el===document.getElementById("app")//->true。

（4）Vue实例化过程中需要经历一系列额初始化过程，如：配置数据观测--编译模板--挂载实例到DOM--数据变化时更新DOM。

**2.vue.extend和vue.component**

（1）extend是构造一个组件的语法器，传入参数，返回一个组件。然后可以使用Vue.component全局注册方法注册这个组件，也可以注册为局部组件。





（2）Vue.component也可以用来注册（全局）组件，如果传入1个选项对象。



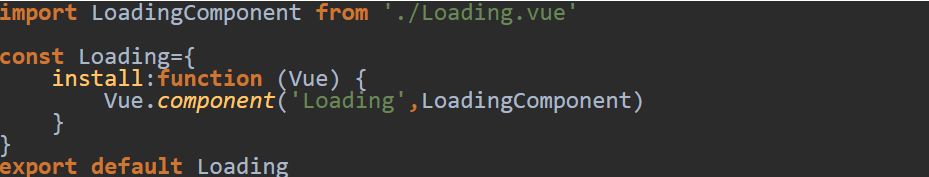


Vue.component和vue.extend基本相同，返回的都是1个组件（组件还没有挂载）。

（3）和Vue.component和vue.extend不同，new Vue会将实例和容器绑定，即，返回的是vue内部不同的类。

**3.插件机制**

（1）如element-ui和axios，前面需要使用Vue.use，而后者需要使用import。原因在于element-ui内有install方法。如果使用Vue.use()方法，则会默认调用install方法。



* + 1. 异步组件和懒加载

**1.异步组件**

（1）在大型应用中，可能需要将应用拆分为多个小模块，按需从服务器下载。

（2）Vue.js允许将组件定义为一个工厂函数，异步地解析组件的定义。只在组件需要渲染时才触发工厂函数，并且把结果缓存起来，用于后面的再次渲染。



当工程函数接收到resolve时，将下载调用组件。



（3）当一个异步组件被作为vue-router的路由组件使用时，这些高级选项都是无效的，因为在路由切换前就会提前加载所需要的异步组件。

（4）异步组件需要结合webpack代码分离实现。

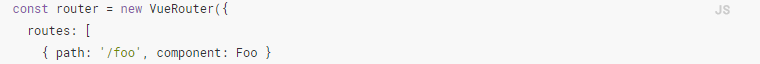
**2.路由懒加载**

（1）需要结合webpack代码分离和vue异步组件。

（2）首先，可以将异步组件定义为返回一个Promise的工厂函数。需要使用动态import语法来定义代码分块点。



（3）在路由配置中什么都不需要改动，即可实现懒加载。



* + 1. 生命周期钩子

（1）所有的生命周期钩子都自动绑定this上下文为实例，因此，钩子内可以访问实例数据。

（2）因此，不能使用箭头函数来定义一个生命周期方法，因为箭头函数绑定了父上下文。

（3）钩子说明：

|  |  |
| --- | --- |
| beforeCreate | 组件实例已创建（new了1个js对象，其他都没有），但还没有绑定属性。 |
| created | 属性已绑定（数据、方法等），但还没有挂载$el，即dom还没有生成。 |
| beforeMount | （1）挂载开始之前被调用：render函数将首次被调用。  （2）服务器端渲染期间不会被调用。 |
| mounted | （1）vm.$el挂载之后调用该钩子。  （2）不会承诺所有的子组件也都一起被挂载。如果希望等到整个视图都渲染完毕，可以用vm.$nextTick替换掉mounted。 |
| beforeUpdate | （1）数据更新时调用，发生在虚拟DOM重新渲染和打补丁之前。  （2）可以在这个钩子中进一步地更改状态，这不会触发附加的重渲染过程。 |
| updated | 组件DOM已经更新。应该避免在此期间更改状态。 |
| 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。 |
| destroyed | Vue实例销毁后调用。 |
| errorCaptured | 捕获一个来自子孙组件的错误时被调用。 |

* + 1. 动画

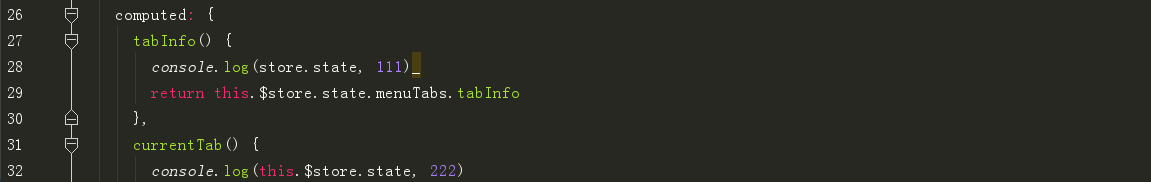
（1）在进入/离开的过渡中，会有6个class切换：v-enter、v-enter-active、v-enter-to等。

（2）CSS动画用法同CSS过渡，区别是在动画中v-enter类名在节点插入DOM后不会立即删除，而是在animationend事件触发时删除。

* + 1. 使用

（1）对于模板中文本使用{{}}语法，但对于动态绑定1个属性，需要使用v-bind。

如下,组件的computed属性中使用了store。如果有模块，需要在state后面加上模块名称。



绑定的事件时，要获取绑定事件元素，要用e.currentTarget。









自定义方法

props

修改对象

router-link的写法，可以直接写：

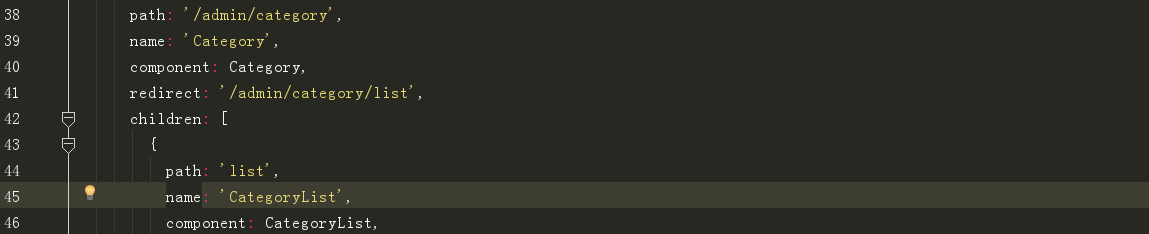


也可以使用绑定：



默认是绝对路径，可以使用append属性，变为相对路径。

默认路由使用redirect，需要使用绝对路径，如下：



（5）数组变异方法可以直接检测到，如push、pop等方法，但数组替换方法需要使用新数组代替旧数组，如filter、concat等方法。



（6）因为js原因，当利用索引直接设置一个项：vm.items[indexOfItem]=newValue，或直接使用length修改1个数组长度时，如，vm.items.length=newLength，vue不会检测到。

对于第1个问题，可以使用如下方法解决：





对于第2个问题，可以使用如下方法解决：



（7）因为js原因，Vue不能检测对象属性的添加或删除。可以使用如下方法解决：





（8）为已有对象赋予多个新属性，应该如下方式：

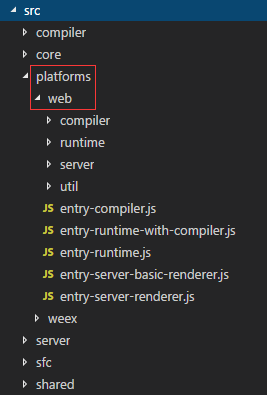
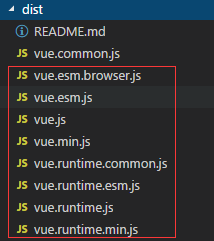


（10）如果需要显示一个数组的过滤或排序副本，而不实际改变或重置原始数据，可以创建返回过滤或排序数组的计算属性。

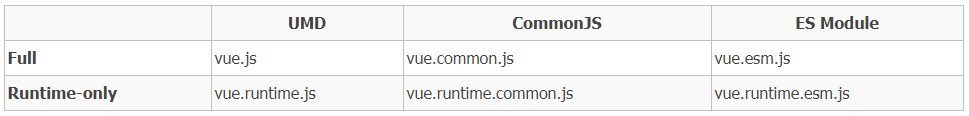
1. 源码
   1. 准备

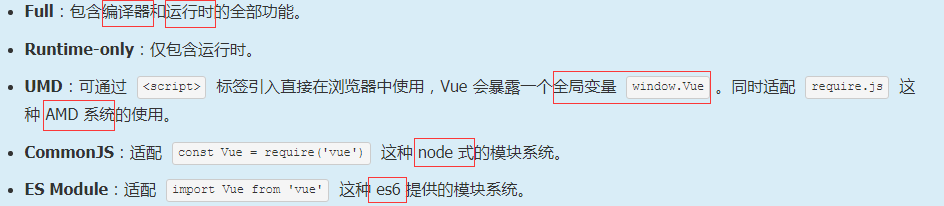
**1.入口文件**

查看package.json,入口文件就是src/platforms/web下的entry-\*系列文件。

（1）编译后的出口文件区别



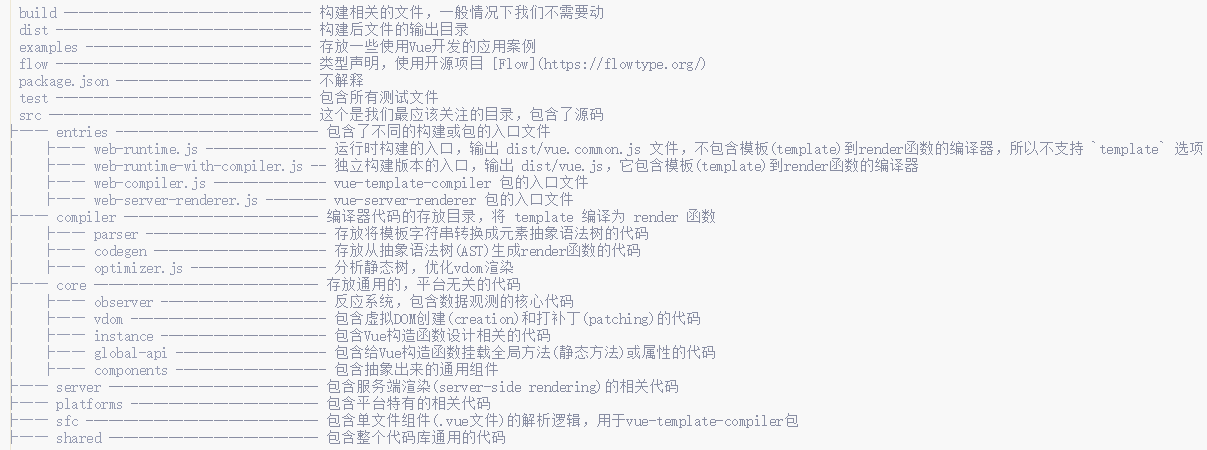


如果需要使用Vue提供的html模板功能，就使用Full版本。否则，最好用Runtime-only版本，因为它比Full版本的文件体积小30%。\*.vue单文件组件会被vue-loader或vueify直接构建成JavaScript，并没有使用到Vue的编译器，因此可使用Runtime-only版本。

（2）目录文件区别

之所以要将Vue多次赋值到\_base、proxy等，目的是保存具有不同属性的Vue，因为它们保存的地方不同。

与platform同级的core、compiler、server等目录，应该是web和weex共用的文件。



**2.Vue定义流程**

（1）以src/platforms/web-runtime-with-compiler.js为例，它从src/platforms/web/runtime/index.js引入Vue，并给Vue添加原型方法$mount（将之前的$mount缓存，使用新的方法覆盖，如果执行的话也会执行之前的那个mount）和静态方法compile。

（2）src/platforms/web/runtime/index.js从src/core/index.js引入Vue，对其静态属性config和options增加一些属性，也添加了1个原型方法$mount。注意这个$mount方法和web-runtime-with-compiler.js不同，应该没有编译器功能。

（3）src/core/index.js从src/core/instance/index.js引入Vue，有以下功能：

通过initGlobalAPI方法给Vue添加：1）config；2）util；3）set、delete、nextTick；4）options；5）use方法、mixin方法、extend方法、initAssetRegisters。

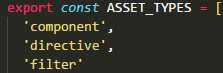
添加属性FunctionalRenderContext。

（4）在src/core/instance/index.js中，才是Vue构造函数真正定义的地方。定义非常简单，内部就执行1个\_init方法。

文件中还通过initMixin、stateMixin、eventsMixin、lifecycleMixin、renderMixin给Vue添加\_init，$watch，$on、$once、$off、$emit，\_update、$forceUpdate、$destroy，$nextTick、\_render等原型方法。

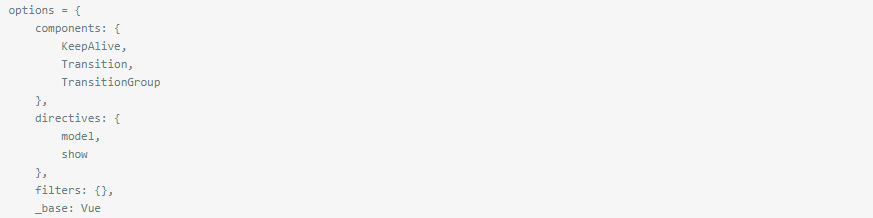
**3.初始化配置**

在使用Vue之前，会做如下初始化配置：

（1）在core/index.js的时，会将初始化。

（2）添加\_base和内置组件到配置中。

（3）初始化平台特定的指令和组件。

此时的options为：。

* 1. 生命周期
     1. 配置处理

**2.合并配置**

实例化时，就会在\_init中合并定义组件的配置。

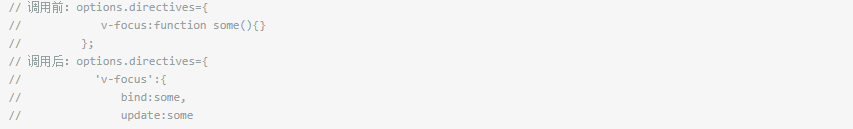
（1）是实例化组件时使用，暂时不考虑。

（2）在mergeOptions前，需要先执行resolveConstructorOptions，不过它直接返回Vue.options。

（3）具体合并前，还会对配置检查和格式化。首先检查配置名称是否与保留名称冲突。然后使用normalizeProps转为驼峰形式，并设置type类型。



然后使用normalizeDirectives将directives格式化如下：

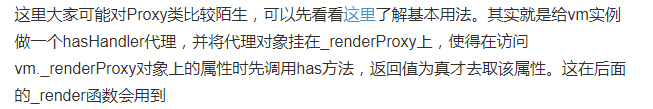


（4）正式合并：mergeOptions用于父子选项值（Vue自身和传入的）。

* + 1. 初始化实例

（1）添加属性

添加\_renderProxy和\_self属性，值都为vm自身。



（2）初始化生命周期initLifecycle

主要初始化parent、children等属性。

（3）初始化事件系统initEvents

会创建属性\_events等。

（4）初始化编译函数initRender

包括创建属性\_vnode、slots、scopedSlots，添加方法createElement、\_c等，还会添加属性$attrs、$listeners。

（5）执行生命周期钩子beforeCreate

（6）初始化插件系统initInjections

（7）将props、methods、data、computed、watch转为响应式initState

（8）初始化provide。

（9）执行生命周期钩子created函数

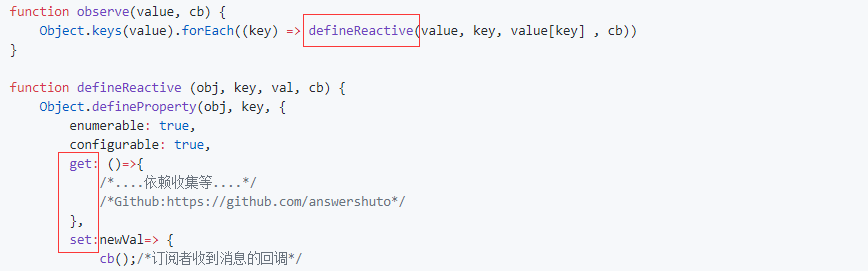
（10）调用mount函数挂载

* 1. 响应式原理
     1. 概述

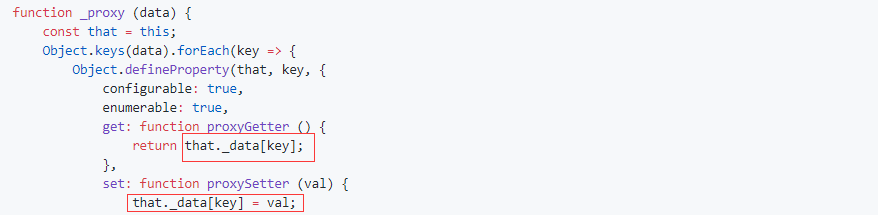
Vue通过如下方式和步骤实现响应式：

（1）通过Object.defineProperty将data变为可观察对象。





（2）将\_data上的数据代理到vm实例上，目的是便于访问。



（3）通过“发布者—订阅者”模式，使用Watcher作为数据订阅者，使用Dep作为数据收集者，收集订阅者。

* + 1. 实现

（1）在state.js的initData中将数据进行Observer。

在initMixin->initState中调用。

首先检查data属性是否为函数，是则获取data。然后逐一检查data中的属性和props、methods中的属性名是否有冲突。最后调用observe方法转化。

observe方法位于src/observe/index.js文件内。注意同一文件下还有Observer类。

尽管observe的返回值在这里没有被使用，但因为引用，所以还是起作用的。



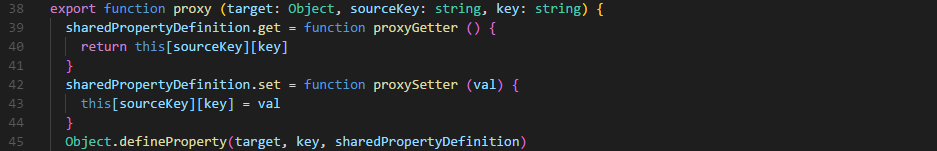




（2）代理

在initData中，使用vm的属性代理\_data的属性。





（3）订阅者和收集者

以计算属性为例，如果有计算属性，会在initState中调用initComputed，其中就会实例化订阅者Watcher。

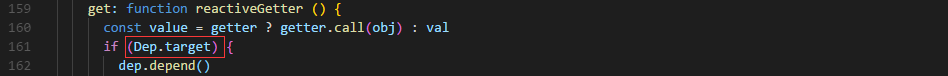


可见，每个需要观察的值，都会实例化1个Watcher。

如何将订阅者收集，当数据变化时，只通知那些相关的订阅者？这里很巧妙，先触发1个data的get。



在get中，先调用pushTarget方法，将Watcher保存到Dep.target中（注意pushTarget在src/observe/dep.js文件中），然后调用Watcher的getter方法，就是获取它依赖的值，此时就回到了src/observe/index.js文件的get观察方法中。



因为Dep.target有值，就会执行dep.depend。最后会调用Dep的addSub方法，将watcher保存到Dep的subs属性中。



而且，在observe中可以看到，每个被观察的属性，都会实例化1个Dep。注意这里是两个过程，getter哪里已经结束了，但Dep的实例保留在作用域下。这里调用getter，使用的就是那个作用域下的Dep。



* + 1. 其他问题

**1.对象属性的递归观察**

**2.数组的特殊处理**

* 1. 模板编译
     1. 概述

**1.$mount方法**

入口文件中，entry-runtime.js用于构建仅包含运行时的源码文件，而entry-runtime-with-compiler.js用于构建同时包含编译器和运行时的全功能文件。因此，两个文件的差集必然就是编译器实现。

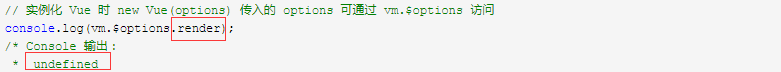
entry-runtime.js中仅有2行代码，而entry-runtime-with-compiler.js主要为Vue的prototype扩展了一个$mount方法。

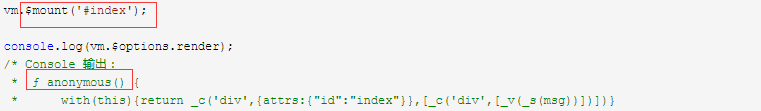
（1）使用场景

$mount方法为vue实例添加1个render方法，render方法可以由如下的html模板文件编译而来。挂载前，vm.$options.render为undefined，之后为1个匿名函数。

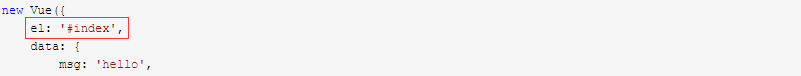






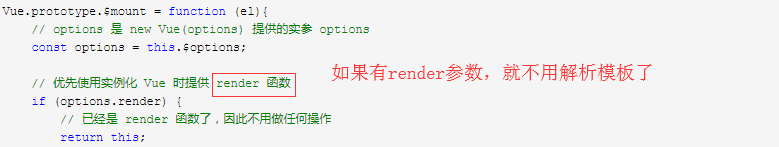


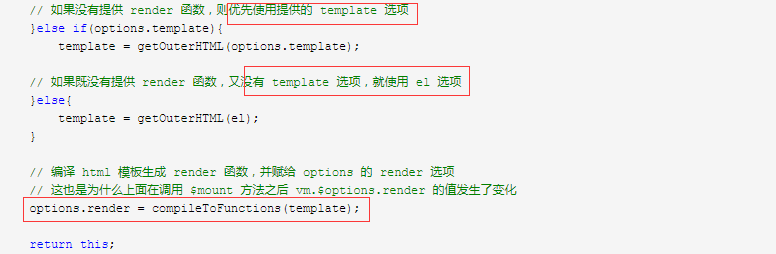
上面的过程和下面的方法等效：

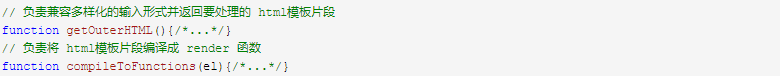


（2）$mount源码分析（伪代码）

定义在Vue的prototype上，使得每一个Vue实例都能使用$mount方法。

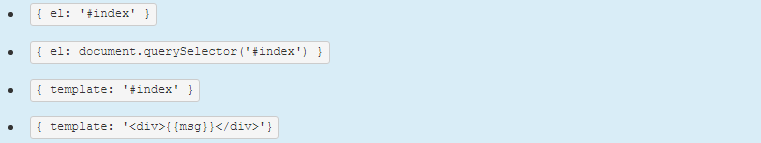






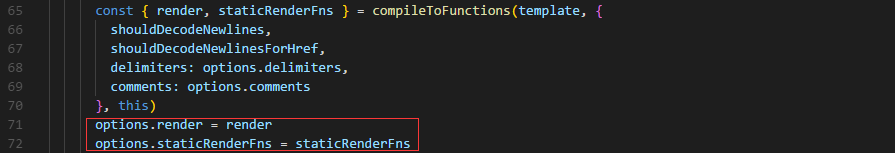
由此可见：如果同时提供render、template、el选项，使用的优先级是render>template>el。

其中，getOuterHTML函数作用是兼容各种使用场景，返回值是DOM的outerHTML，负责得到html模板片段。



**2.compileToFunctions方法**

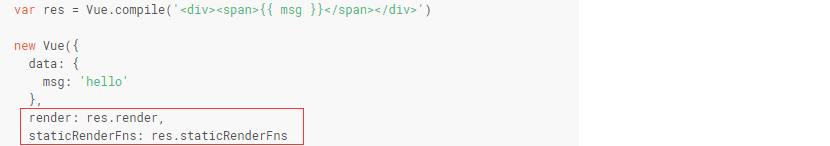
从$mount代码可见，真正将模板编译为render函数的，是compileToFunctions方法。真正的代码如下：



在entry-runtime-with-compiler.js中，有如下代码：



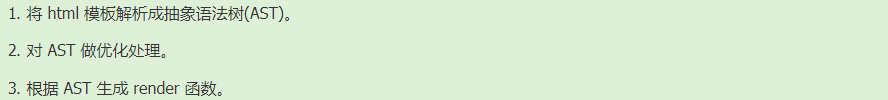
即，Vue.compile(template)这个API就是compileToFunctions。API文档中对Vue.compile有如下说明：

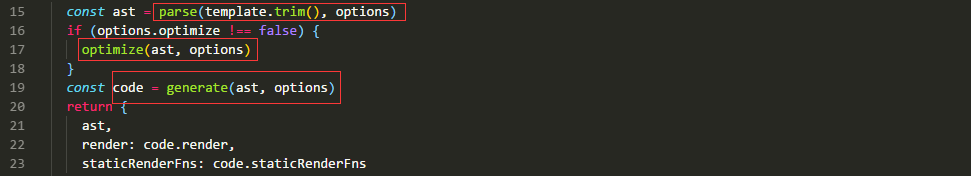






compileToFunctions由createCompiler方法生成，createCompiler在src/compile/index.js文件中。具体执行代码的过程（编译模板）主要分为三步：





* + 1. 生成AST

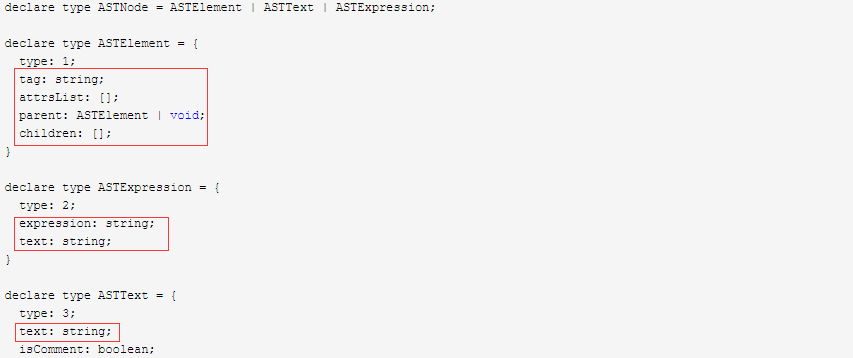
**1.AST节点类型**

Vue的html模板比较特殊，它算不上是一门语言，而是基于HTML的声明式绑定。因此，Vue生成的AST类似于DOM树。特别注意的是，Vue编译器的编译结果是一个函数——Vue的render函数，AST只是方便处理的中间形式。

代码位置：src/compiler/parser/html-parser.js。

Vue的AST节点数据结构有3种，定义位置为vue/flow/compiler.js，如下：





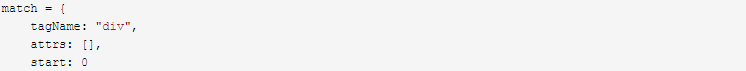
文本节点就是普通的文本，表达式节点就是插值语法解析后的结果。因为它们都是叶子节点，因此没有parent和children字段。

**2.解析过程**

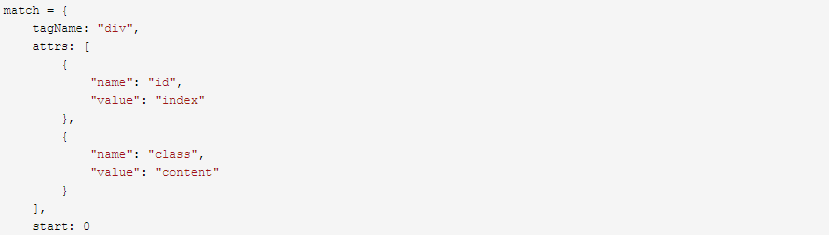
（1）使用parseStartTag函数解析开始标签



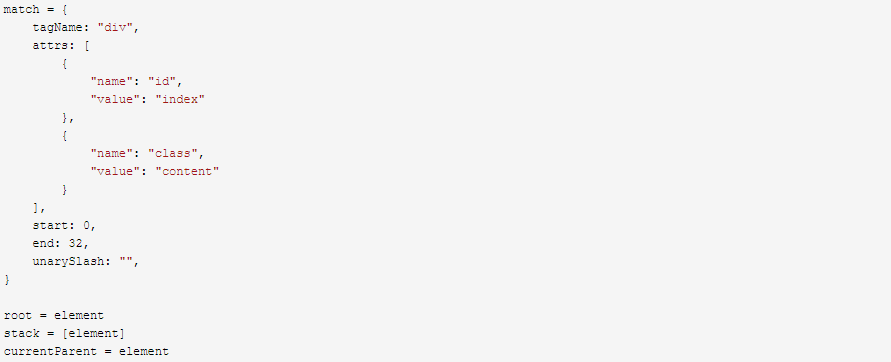
以解析为例，第一步是解析标签名，得到结果如下：



然后通过while循环解析所有的属性键值，得到结果如下：



第三步解析开始标签闭合部分，并生成了一个AST节点，最终的变量状态如下：



（2）使用parseEndTag解析结束标签（前面解析的是开始标签）



因解析开始标签的时候，会将解析到的标签逐个压入stack栈中。在解析结束标签时，会去找栈中离的最近的同类型标签，并更新currentParent。如stack开始为['div','p','a']，经过parseEndTag之后可能就会变成['div','p']，currentParent也从指向a变成了指向栈顶的p。



（3）解析文本

因Vue是支持在文本中插值，如，<div>hello,{{msg}}</div>。文本解析就是解析这些混在文本中的表达式。上面的代码经过文本解析为['"hello"','\_s(msg)']。

（4）parseHTML整体解析过程

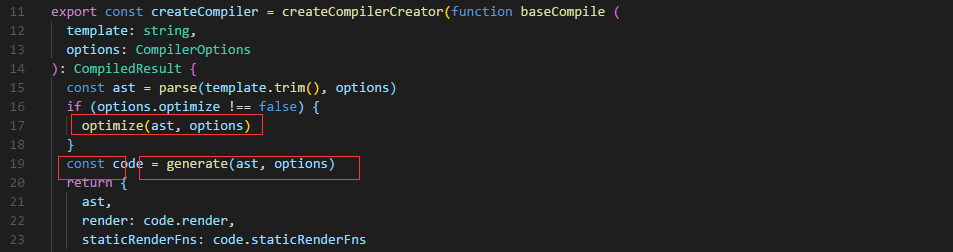


parseHTML是通过循逐步将整段html全部解析完。

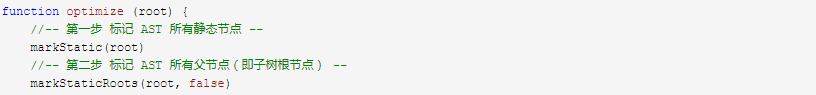
* + 1. 生成render函数

**1.优化代码**

src/compiler/index.js

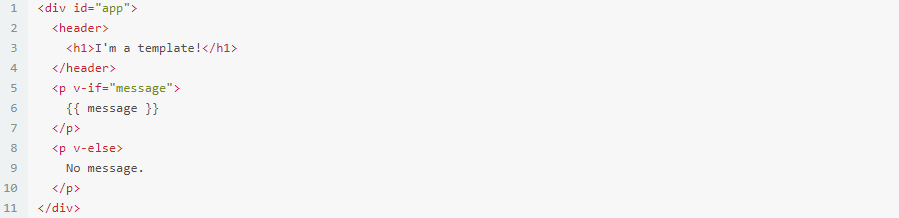


优化原理很简单，就是将那些更新视图时根本不需要改变的纯静态的节点标记为static，在页面重渲染时可直接跳过它们。



**2.生成render函数**

模板文件编译后就得到render函数。其运行结果就是VNode。如下面的template编译出来的render为：





* + 1. staticRenderFns

staticRenderFns数组中的函数与VDOM中的diff算法优化相关，我们会在编译阶段给后面不会发生变化的VNode节点打上static为true的标签，那些被标记为静态节点的VNode就会单独生成staticRenderFns函数。如上面模板中的header就会生成如下：



* + 1. 虚拟dom

**1.流程**

（1）上述过程执行完后，调用entry-runtime.js中的mount，会继续调用web/runtime/index.js中的$mount。其中实际执行的是core/instance/lifecycle.js中的mountComponent方法。

（2）在mountComponent中，会实例1个Watcher，因默认Watcher会立即执行expression，因此会立即执行updateComponent。





（3）\_render方法在core/instance/render.js中定义。核心在于执行之前生成的vm.$options.render，生成1个虚拟dom。



（4）然后在\_update方法中会使用\_\_patch\_\_执行真实dom的创建、虚拟dom的diff修改、dom的销毁、挂载等。vm.\_\_patch\_\_就是元素渲染、vnode做diff并修改、元素销毁的地方。

* 1. 虚拟DOM
     1. 概述

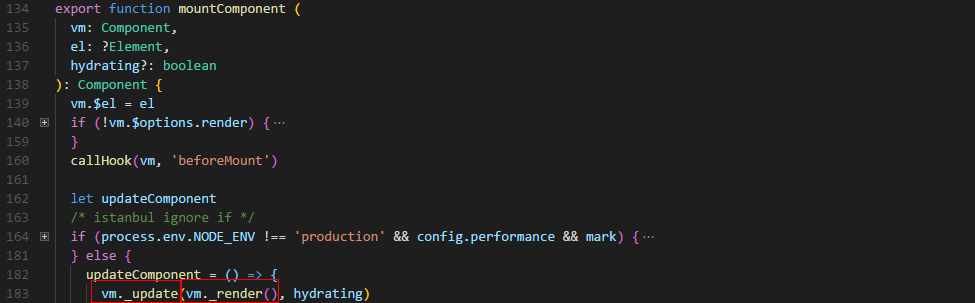
**1.使用虚拟dom的意义**

因原生dom有非常多地属性，为提高性能，需要尽可能多地复用旧的dom（注意，不是减少dom操作，因为之前也是这样做的）。

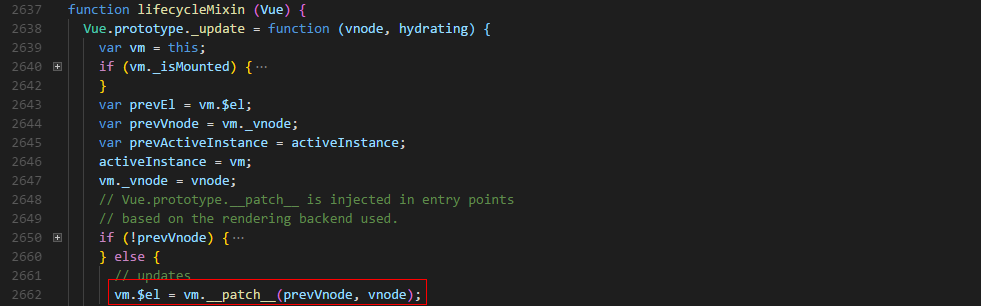
**2.流程**

（1）调用之前生成的render函数，生成虚拟dom——vnode。





（2）使用vm.\_\_patch\_\_方法，执行diff操作，根据最后的vnode，绘制页面。

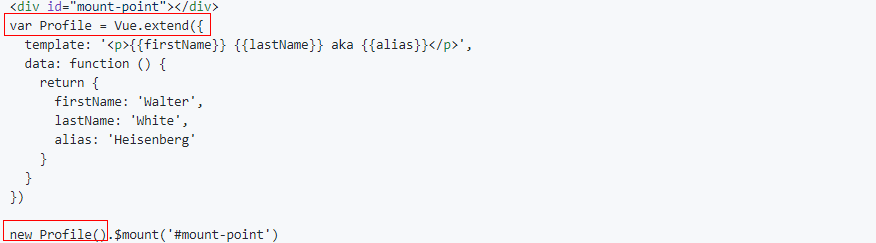






* + 1. Vue.extend

**1.使用**



页面输出（替换div标签）：



**2.作用**

（1）方法定义在src/core/global-api/extend.js中的initExtend方法内。

（2）Vue.extend内部会定义了1个构造函数——VueComponent方法，它和Vue一模一样，内部只执行\_init方法。



Vue.extend最后返回的也是这个方法。



因此，使用上，它和Vue构造函数一模一样。

**3.和Vue构造函数的区别**

（1）因Vue.extend仍然只是Vue的1个方法，因此该方法内部的this，指向Vue的实例，因此命名为Super。



Sub的原型继承Super的原型。



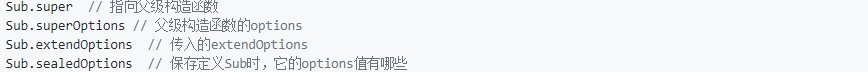
将Vue.extend传入的参数作为extendOptions。



连同Super.options，使用mergeOptions合并作为Sub.options。



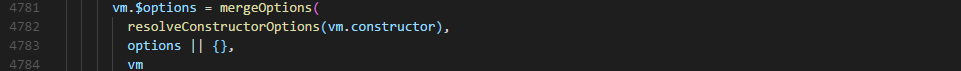
（2）与Vue的构造函数相比，Sub增加了四个全局属性：



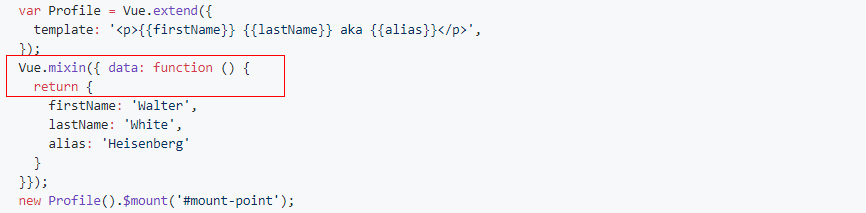
同时也少了一些全局属性：



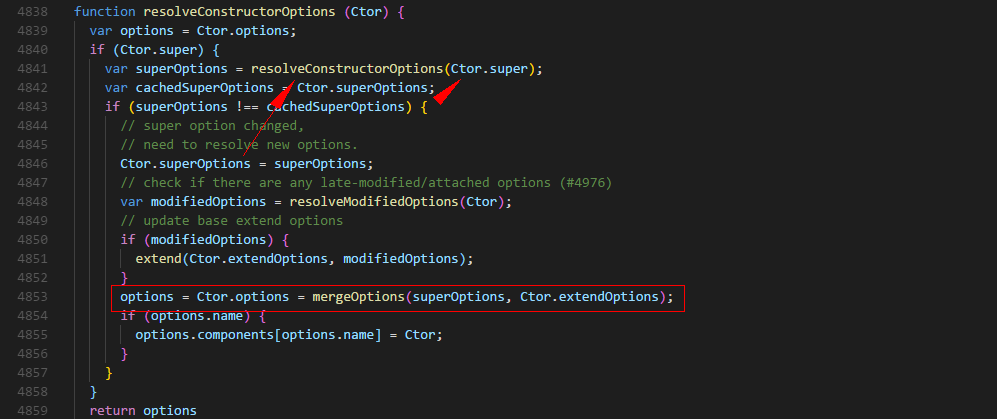
（3）在\_init方法中，调用resolveConstructorOptions合并options时，有：



resolveConstructorOptions中主要考虑的情况就是存在mixin的情况。



此时，Vue（extend的Super）的data属性会发生改变，需要同时改变Vue.extend生成的Sub的superOptions，及其options。



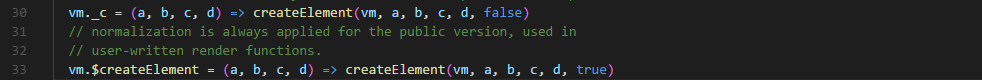
* + 1. VNode

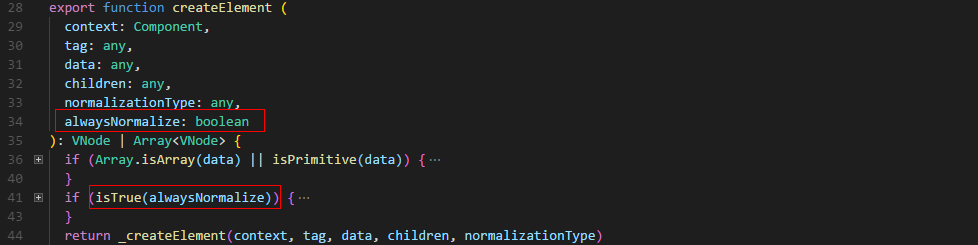
**1.基本构造**

（1）构造函数可以接收七个参数：tag标签名、data结点相关数据、children子结点对象数组、text文本内容、elm原生结点元素、context指当前元素所在的Vue实例、componentOptions保存自定义组件上部分组件属性。



（2）render函数中，内部调用的都是createElement方法。





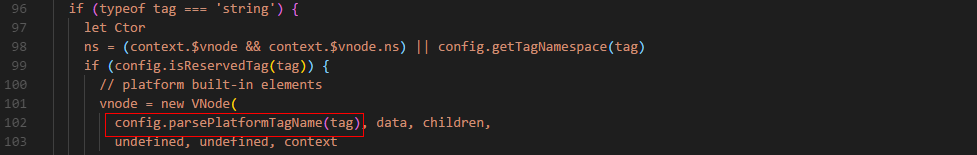
（3）在\_createElement中，有：

如果定义了is，则替换tag为is：



如果tag为空，则返回1个空的VNode。

如果tag是字符串，检查是否是平台保留标签名，是则直接创建VNode对象。



如果tag是字符串，且是components中定义的组件名，则创建组件。



如果tag是字符串，但既不是保留标签名，也不是组件名，则直接创建。



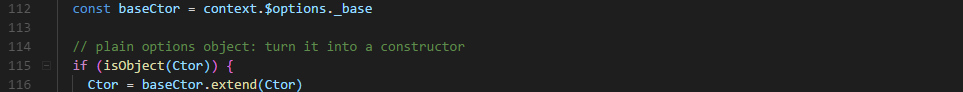
如果tag不是字符串，也使用createComponent创建。



（3）createComponent

createComponent主要处理两种情况：

1）tag为对象，即上面的tag不是字符串，内部调用的是baseCtor.extend。即，如果tag不是字符串，要想成功编译，就必须满足extend的条件。



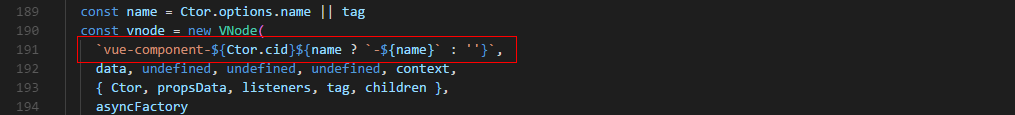


2）自定义组件：则必须是1个函数。其中会把所有的组件属性，保存到data对象中，传入VNode构造函数中。









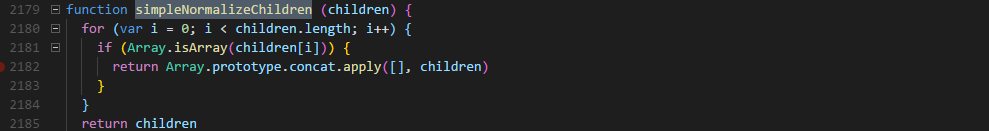
其中还设计函数式组件和异步组件等内容。

* + 1. children的归一化处理

即，将多维的数组，合并转换成一个一维的数组。代码在src/core/vdom/helpers/normalize-children.js中。

（1）不需要归一化处理：对于简单html标签生成的render函数，因返回的是一个VNode数组，可以直接跳过归一化。

（2）简单归一化：处理自定义组件情形。



（3）完全归一化：子元素包含生成嵌套数组的结构，比如<template>、<slot>、v-for或子元素是用户手写的render函数。

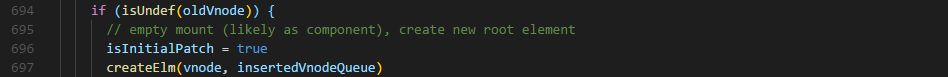
* + 1. 使用\_\_patch\_\_方法执行diff并挂载

**1.流程**

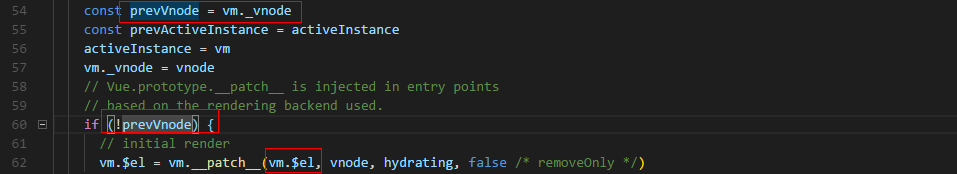
（1）如果vnode未定义，但oldVnode有值，则销毁oldVnode。



（2）如果oldVnode未定义，isInitialPatch置为true，然后调用createElm。



（3）初始化情况：第1次执行时，因为prevVnode不存在，则设置为挂载元素el。



此时，使用emptyNodeAt创建1个el的vnode。然后执行createElm。



（4）如果oldVnode是vnode，且和新生成的vnode是同一节点，则执行patchVnode，即diff。

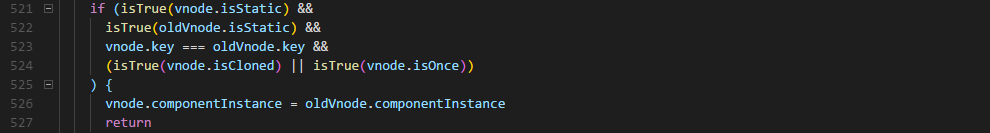


**2.patchVnode**

目的是尽可能复用dom元素。

（1）如果oldVnode和vnode是同一个对象，则直接返回。

（2）如果当前节点是静态内容节点、key相同的节点、once节点，或isCloned节点，则直接用oldVnode的elm和componentInstance替换vnode上的相同属性。



（3）其他节点，则需要处理子节点。



如果新节点的子节点不是文本节点，当新、旧节点都有子节点，执行diff算法。



如果旧节点没有，只需要将新的添加即可。如果新的没有，只需要将旧的移除即可。

如果新节点是文本节点，则直接修改文本内容。

**3.diff算法**

（1）没有key的情况：

假设oldCh上有五个元素a、b、c、d、e，newCh有六个元素d、e、b、f、d、a，且没有key值。初始情况下，页面中dom顺序为abcde。

第一次while循环，oldStartVnode和newEndVnode都是a。此时,a元素会把插入到e元素的下一个元素前。此时页面中dom变为bcdea，oldStartVnode指向b，newEndVnode指向d。（[b]cd[e]a，[d]ebf[d]a）

第二次while循环，因为头尾都不相同。此时，会创建新的元素d并插入到b前面。此时页面中dom变为dbcdea，newStartVnode指向e。（d[b]cd[e]a，d[e]bf[d]a）

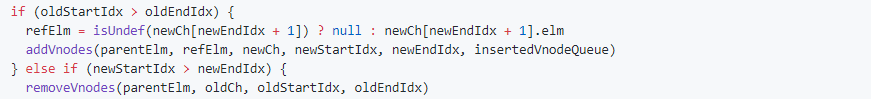
第三次while循环，oldEndVnode和newStartVnode都是e。此时，e元素会插入到b元素之前。此时页面中dom变为debcda，oldEndVnode指向d，newStartVnode指向b。（de[b]c[d]ea，de[b]f[d]a）

第四次while循环，oldStartVnode和newStartVnode都指向b，直接复用b元素。此时，oldStartVnode指向c，newStartVnode指向f。（deb[c][d]ea，deb[f][d]a）

第五次while循环，oldEndVnode和newEndVnode都指向d，直接复用d元素。此时，oldEndVnode指向c，newEndVnode指向f。（deb[c]dea，deb[f]da）

第六次while循环，两个数组中都只剩下一个没有遍历的元素且不相同。因此，后创建新的元素f并插入到c前面。此时页面中dom变为debfcda，newStartVnode指向b。

这时newStartIdx会大于newEndIdx，所以会终止循环。这时我们发现，页面中多了c元素。所以updateChildren方法在循环之后还有删除无用的旧结点的操作。



（2）有key的情况：

假设oldCh上有五个元素a、div[key=1]、footer[key=3]、span[key=2]、p。newCh有六个元素p[key=3]、span[key=2]、p、div[key=1]、a、span。初始情况下，页面中dom顺序为a、div[key=1]、footer[key=3]、span[key=2]、p。

第一次while循环，头尾都不可复用，所以此时会生成oldKeyToIdx。



newStartVnode元素p[key=3]根据key值比较，elmToMove会指向footer[key=3]，但因为它们标签名不一样，所以sameVNode判断会返回false。所以直接插入到a前面，页面中dom变为p[key=3]、a、div[key=1]、footer[key=3]、span[key=2]、p，newStartVnode指向span[key=2]。

第二次while循环，同样头尾都不可复用。此时，newStartVnode元素span[key=2]根据key值比较，elmToMove会指向span[k=2]，两元素可以复用，span[k=2]会被插入到a前面，页面中dom变为p[key=1]、span[key=2]、a、div[key=1]、footer[k=3]、p，newStartVnode指向div。同时oldCh变为[a, div[key=1], footer[k=3], undefined, p]。

第三次while循环，oldEndVnode和newStartVnode都是p。此时，dom中最后的p元素会插入到a元素，页面中dom顺序变为p[key=1]、span[key=2]、p、a、div[key=1]、footer[k=3]，oldEndVnode前移一位指向了undefined，newStartVnode后移一位指向div[key=1]。

第四次while循环，oldEndVnode返回undefined。页面中dom不变，oldEndVnode指向footer[k=3]。

第五次while循环，依然头尾都不可复用。newStartVnode根据key=1找到可以复用的div[key=1]，该元素会插入到a元素之前，页面中的dom变为p[key=1]、span[key=2]、p、div[key=1]、a、footer[k=3]，同时oldCh变为[a, undefined, footer[k=3], undefined, p]，，newStartVnode后移一位指向a`。

第六次while循环，newStartVnode和oldStartVnode都指向a。页面中dom不变，newStartVnode后移一位指向span，oldStartVnode后移一位指向undefined。

第七次while循环，oldStartVnode返回undefined。页面中dom不变，oldStartVnode指向footer[k=3]。

第八次while循环，新旧没有比较的子元素都只剩一个，且不可复用，页面中会创建span，并插入到footer[k=3]之前。此时页面中dom变为p[key=1]、span[key=2]、p、div[key=1]、a、span、footer[k=3]，newStartVnode指向超出newCh范围，指向undefined。

这时newStartIdx会大于newEndIdx，所以会终止循环。同时会删除多余的footer[k=3]。

**3.createElm**

（1）如果是组件，则执行createComponent。

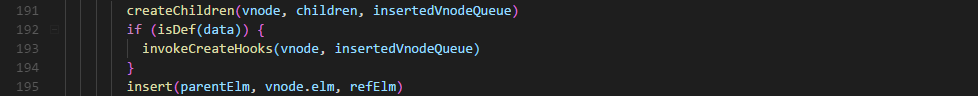
（2）如果不是组件，当定义了tag时，使用createElementNS或createElement根据vnode创建真是element。



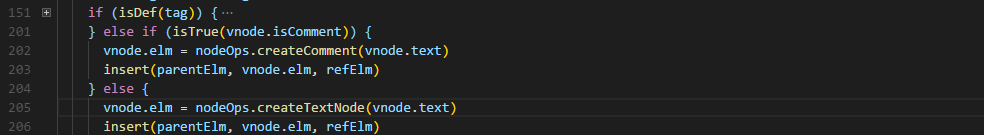
使用setScope设置scoped CSS的属性。



使用createChildren创建子节点，内部也是调用createElement方法。最后调用insert插入到页面中。



（3）如果为注释或文本节点，创建对应dom并插入。

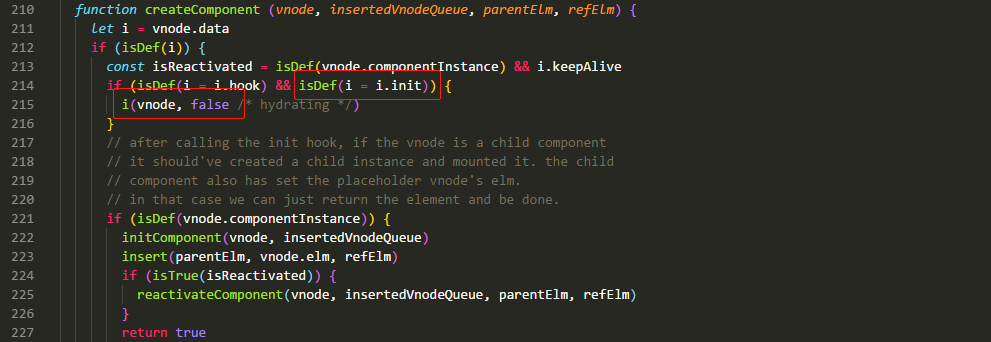


* + 1. 自定义组件

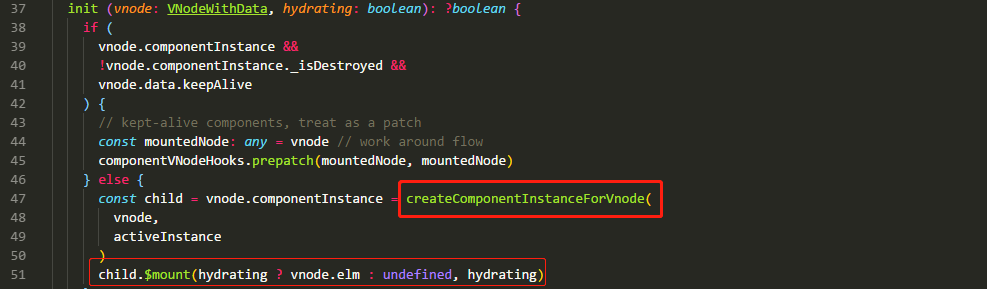
**1.4个钩子函数**

在create-component.js中，定义了4个钩子函数：init、prepatch、insert、destroy。

在调用createComponent创建自定义组件过程如下：



init就是第1个钩子函数。



createComponentInstanceForVnode方法中，会调用组件的构造方法，创建新的Vue实例。实例返回到child中，然后挂载。

**2.prepatch**

prepatch在进行diff操作之前执行，前提是旧新Vnode实例相同。

* 1. 其他
     1. mergeOptions：配置合并策略

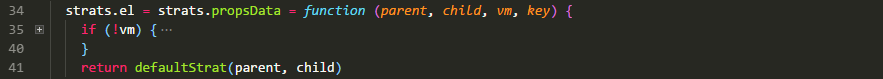
主要做了这么几件事（在src/core/util/options.js中）：

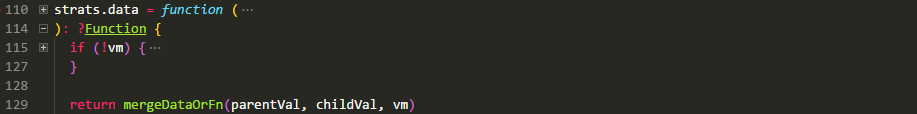
1）定义strats为1个空对象。





2）在strats对象上定义与参数选项名称相同的方法。

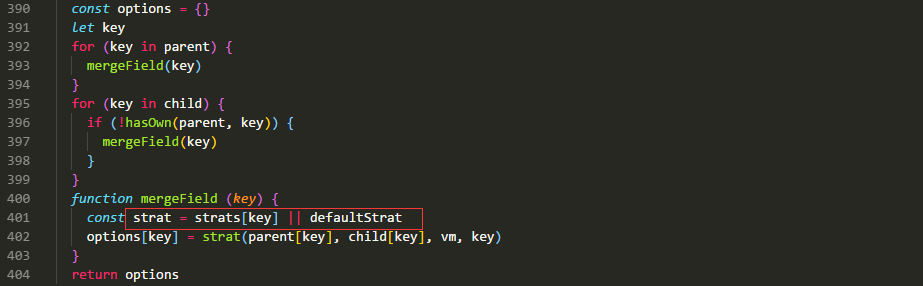




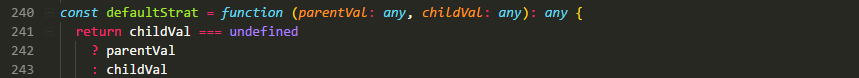




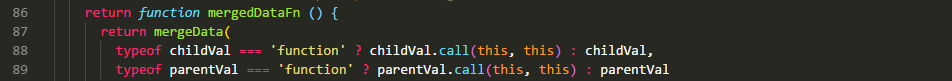
3）根据参数选项调用同名的策略方法进行合并处理。



（4）对于el和propsData选项，合并策略为优先子元素。



对于data，返回的是1个函数，函数内会将父子data深度合并（如果data是函数，就先求值，然后在深度合并）。



对于10个生命周期函数，将和mixins中同名属性使用数组保存。mixins中的优先级高，先调用。



对于3个assets（directives、components、filters），使用继承策略合并，以parent中同名属性的值为原型对象，以child中同名属性的值做实例属性。



watch同生命周期函数的合并策略。props、methods、computed使用覆盖策略，mixins中的同名属性会被child覆盖。

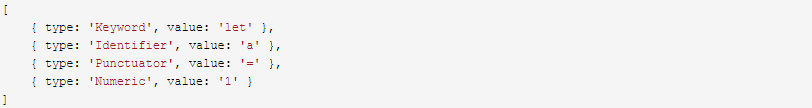
* + 1. 构建AST的一般过程

抽象语法树(Abstract Syntax Tree)是源代码语法结构的抽象表示，并以树这种数据结构进行描述。Vue同样也是使用AST作为中间形式完成对html模板的编译。

通常，程序语言转为AST会分为两步：词法分析和语法分析。如下面的js语句：



词法分析器会把代码的字符序列转换为单词序列（tokens），可以得到如下一个词素列表：



语法分析器会在词法分析的基础上将单词序列（tokens）组合成各类语法短语（语句、表达式等），之后可以得到AST的JSON格式：



上面就是使用Esprima工具对JS代码进行词法分析和语法分析后的结果。可以使用http://astexplorer.net/和http://resources.jointjs.com/demos/javascript-ast在线生成ast和查看树形工具。

* + 1. 工具函数等

**1.shared/util.js**

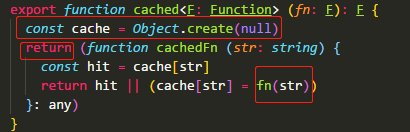
内部为跨平台用到的工具函数。

（1）makeMap

用于isBuiltInTag、isReservedAttribute等。将1个字符串解析为map，返回1个函数，判断是否在map中。map中元素的值为true。

（2）cached

使用了闭包。将1个函数的执行结果保存在闭包中的cache中。



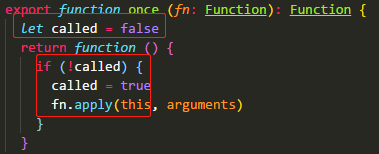
（3）noop为不执行任何操作的空函数。

（4）looseEqual

判断两个值是否是普通意义上的值相等，如111和'111'相等。如果是对象，会使用递归判断。

（5）once

也是使用闭包特性实现。



**2.sfc/parser.js**

只暴露parseComponent函数，用于将单文件组件（.vue）转化成一个sfc对象（可识别的组件对象）。最终的vue.js中不会包含这一部分。

（1）内部先定义sfc对象，以及start、end方法，然后调用compiler/parser/html-parser中的parseHTML，使用start和end方法解析。

（2）start函数用来确定特殊标签<script>,<style>,<template>的开始，并将其内容信息存储到sfc上。其中，script和template直接添加到sfc对象上，style为1个数组，所以可以有多个style。

（3）每当接受到1个结束标签，就会执行end函数。只有当depth=1时，才解析到currentBlock（外部变量，挂载到sfc上）。解析方式也很简单，就是获取lang、scoped、module、src等特殊属性，使用第三方库deindent处理缩进、开始标签的结束位置（>）是否添加换行（用于eslint）。

**3.core/config.js**

Vue的全局配置。首先定义的是1个类型限制，后面默认导出的是这个类型的实现。

* + 1. core/util

**1.debug.js**

（1）前提是非生产环境。

（2）formatComponentName：返回组件名称，可能的值有根实例、有名字的组件、匿名组件。如果是跟实例，，可根据$root是其自己判断。

（3）generateComponentTrace：通过vm.$parent迭代生成1颗从当前组件到跟组件的trace，

（4）warn：在generateComponentTrace生成的trace基础上，打印错误信息。如果有自定义warnHandler，则调用warnHandler。

（5）tip：和warn差不多，不会调用warnHandler。

**2.env.js**

定义用于检测环境的变量，用于兼容。变量有：UA、isIE、inBrowser、isIOS、inWeex等。还定义：

（1）检测浏览器是否支持passive：



（2）定义集合\_Set。

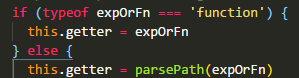
**3.error.js**

导出handleError。和debug.js不同，handleError用于代码中try-catch部分，如执行自定义的钩子函数时。而warn等则用于不符合vue规则的地方。

如果有自定义的$options.errorCaptured，则用，否则用自带的globalHandleError，内部也是调用warn。

**4.lang.js**

定义3个函数：isReserved：判断字符串是否有\_和$开头；def：Object.defineProperty的简写；parsePath解析对象路径中的值，用于core/observer/watcher.js中，将表达式转化为getter函数。

前提是对象key必须有.、$或非字母。

**5.next-tick.js**