Vue-基础篇

1. 指令

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

1.1 概念

1.1.1 参数

v-xxx:参数="var"

- 如： `v-bind:href="url"`

1.1.2 修饰符

修饰符 (Modifiers) 是以半角句号 `.` 指明的特殊后缀

- 包含：.stop、.prevent、.capture、.self、.once、.passive
  - 不像其它只能对原生的 DOM 事件起作用的修饰符， `.once` 修饰符还能被用到自定义的组件事件上
  - 不要把 `.passive` 和 `.prevent` 一起使用，因为 `.prevent` 将会被忽略，同时浏览器可能会向你展示一个警告。请记住， `.passive` 会告诉浏览器你*不想*阻止事件的默认行为。
- 如：<*form v-on:submit* `.prevent` ="*onSubmit*">...</*form*>
- 使用修饰符时，顺序很重要；相应的代码会以同样的顺序产生。因此，用 `v-on:click.prevent.self` 会阻止**所有的点击**，而 `v-on:click.self.prevent` 只会阻止对元素自身的点击。

1.1.3 按键修饰符

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

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

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

- 别名包括： `.enter` 、 `.tab` 、 `.delete` (捕获“删除”和“退格”键)、 `.esc` 、 `.space` 、 `.up` 、 `.down` 、 `.left` 、 `.right` 、 `.ctrl` 、 `.alt` 、 `.shift` 、 `.meta` （win键）
- 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
```

- 你也可直接将 `KeyboardEvent.key` 暴露的任意有效按键名转换为 kebab-case 来作为修饰符：

```
<input @keyup.page-down="onPageDown">
```

- 复合键

```
<!-- Alt + C -->
```

```
<input @keyup.alt.67="clear">
```

- `.exact` 修饰符

  `.exact` 修饰符允许你控制由精确的系统修饰符组合触发的事件

```
<!-- 即使 ctrl+Alt 或 ctrl+Shift 被一同按下时也会触发 -->
<button @click.ctrl="onClick">A</button>

<!-- 有且只有 Ctrl 被按下的时候才触发 -->
<button @click.ctrl.exact="onCtrlClick">A</button>

<!-- 没有任何系统修饰符被按下的时候才触发 -->
<button @click.exact="onClick">A</button>
```

- **鼠标修饰符**

```
<button title="anniu" @mouseup.left="mouseupLeft(a)">测试</button>
```

  - `.left`
  - `.right`
  - `.middle`

1.2 具体指令

- v-bind
  - 缩写：`:name="varName"` 等价于*v-bind:name="varName"*
- v-if
- v-for

```
<li v-for="todo in todos">
      {{ todo.text }}
</li>

<!-- 结果：1 2 3 4 5 6 7 8 9 10 -->
<span v-for="n in 10">{{ n }} </span>

<!-- 也可获取元素的索引，从0开始 -->
<li v-for="(item, index) in ['name', 'age', 'sex']">
     {{ item }} - {{ index }}
</li>

<!-- 迭代对象（object）的属性 -->
<li v-for="value in object">
    {{ value }}
</li>
<div v-for="(value, key) in object">
  {{ key }}: {{ value }}
</div>
<div v-for="(value, key, index) in object">
  {{ index }}. {{ key }}: {{ value }}
</div>
```

> 在遍历对象时，是按 `Object.keys()` 的结果遍历，但是不能保证它的结果在不同的 JavaScript 引擎下是一致的。

- v-for中的元素复用

  > 当 Vue.js 用 `v-for` 正在更新已渲染过的元素列表时，它默认用“就地复用”策略。如果数据项的顺序被改变，Vue 将不会移动 DOM 元素来匹配数据项的顺序， 而是简单复用此处每个元素，并且确保它在特定索引下显示已被渲染过的每个元素

  建议尽可能在使用 `v-for` 时提供 `key`

-
- `v-on:click`
  - 缩写: `@click`
- v-model
  - 表单输入和应用状态之间的双向绑定
  - 多个复选框，绑定到同一个数组：

    ```
    <div id='example-3'>
      <input type="checkbox" id="jack" value="Jack" v-model="checkedNames">
      <label for="jack">Jack</label>
      <input type="checkbox" id="john" value="John" v-model="checkedNames">
      <label for="john">John</label>
      <input type="checkbox" id="mike" value="Mike" v-model="checkedNames">
      <label for="mike">Mike</label>
      <br>
      <span>Checked names: {{ checkedNames }}</span>
    </div>
    ```

    ```
    new Vue({
      el: '#example-3',
      data: {
        checkedNames: []
      }
    })
    ```

  - 类似还有单选按钮、选择框
  - 可以通过 `v-bind` 绑定单选、复选、选择框触发之后的值

    复选：

    ```
    <!-- 默认true/false -->
    <input
      type="checkbox"
      v-model="toggle"
      true-value="yes"
      false-value="no">
    ```

    ```
    // 当选中时
    vm.toggle === 'yes'
    // 当没有选中时
    vm.toggle === 'no'
    ```

    单选：

    ```
    <input type="radio" v-model="pick" v-bind:value="a">
    ```

```
// 当选中时
vm.pick === vm.a
```

选择框：

```
<select v-model="selected">
    <!-- 内联对象字面量 -->
  <option v-bind:value="{ number: 123 }">123</option>
</select>
```

```
// 当选中时
typeof vm.selected // => 'object'
vm.selected.number // => 123
```

- `.lazy` - 抑制 `input` 的事件，变成 `change` 触发

```
<!-- 在“change”时而非“input”时更新 -->
<input v-model.lazy="msg" >
```

- `.number` - 自动转换类型为number，如： `v-model.number="age"`
- `.trim` - 自动去除首尾空白符
-

- v-once - 绑定一次
- v-html
  - 双大括号会将数据解释为普通文本，而非 HTML 代码。为了输出真正的 HTML，你需要使用 `v-html` 指令
  - 不能使用 `v-html` 来复合局部模板，因为 Vue 不是基于字符串的模板引擎

> 你的站点上动态渲染的任意 HTML 可能会非常危险，因为它很容易导致 XSS 攻击。请只对可信内容使用 HTML 插值，**绝不要** 对用户提供的内容使用插值。

- Mustache(双大括号)
  - Mustache(双大括号)语法不能很好地支持布尔类型，只会判断变量存在与否来决定true/false，这种情况应该使用 `v-bind`
  - `Mustache` 和 `v-bind` 中都可以使用js表达式，如： `{{ number + 1 }}`

> 模板表达式都被放在沙盒中，只能访问全局变量的一个白名单，如 `Math` 和 `Date` 。你不应该在模板表达式中试图访问用户定义的全局变量。

- **v-pre**

  跳过这个元素和它的子元素的编译过程。可以用来显示原始 Mustache 标签。跳过大量没有指令的节点会加快编译

```
<span v-pre>{{ this will not be compiled }}</span>
```

- **v-cloak**
-

2. 方法与概念

- 定义模板 - Vue.component

```
// 定义名为 todo-item 的新组件
Vue.component('todo-item', {
  template: '<li>这是个待办项</li>'
})
```

- 数据传递：父→子

  子：

```
Vue.component('todo-item', {
  // todo-item 组件现在接受一个
  // "prop"，类似于一个自定义特性。
  // 这个 prop 名为 todo。
  props: ['todo'],
  template: '<li>{{ todo.text }}</li>'
})
```

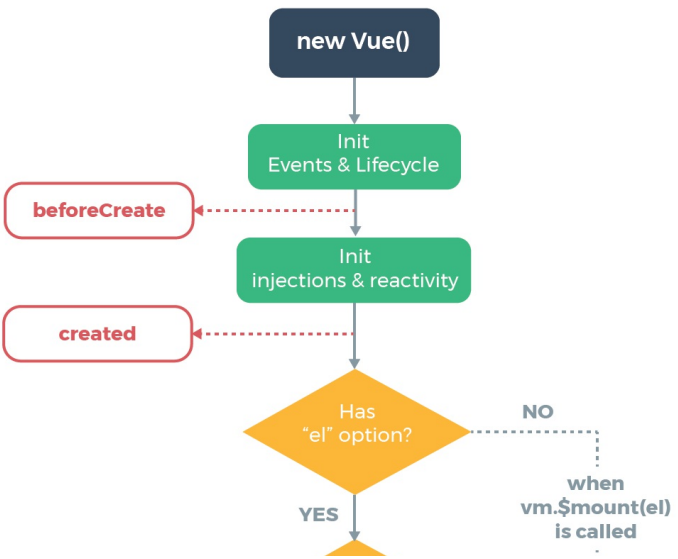
  父：

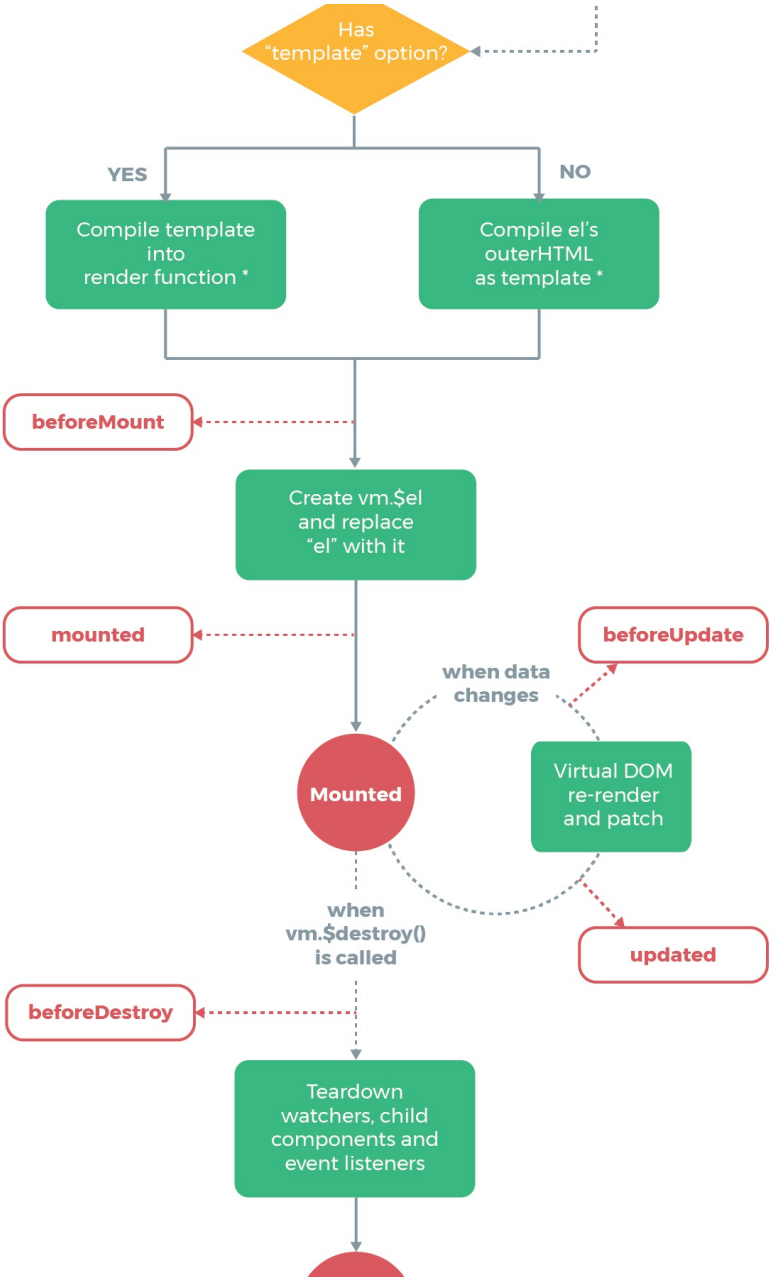
```
<todo-item
      v-for="item in groceryList"
      v-bind:todo="item"    // 这里的todo与子组件中的props中的通道对应
      v-bind:key="item.id">
</todo-item>
```

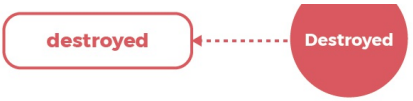
- `$` 和 `_` 获取vue属性

  如：app.$data == app._data

- 声明周期







* template compilation is performed ahead-of-time if using a build step, e.g. single-file components

- **计算属性**

  关键词：`computed`

  目的：避免在模板中出现太多逻辑代码

  特性：当vm.data属性发生变化时，计算属性会随之变化

```
var vm = new Vue({
  el: '#example',
  data: {
    message: 'Hello'
  },
  computed: {
    // 计算属性的 getter
    reversedMessage: function () {
      // `this` 指向 vm 实例
      return this.message.split('').reverse().join('')
    }
  }
})
```

  - 计算属性缓存 vs 方法

    1. 计算属性是计算后缓存起来，只有在依赖发生变化时才再次计算；而函数是每次引用都会计算，无论依赖有无变化；(当计算很消耗性能的时候，缓存起来明显比较好)

    2. 但也要注意：

```
computed: {
  now: function () { // 计算属性now将不再更新，因为 Date.now()不是响应式依赖
    return Date.now()
  }
}
```

  - 计算属性 vs 侦听属性

  - 计算属性的 setter

    实际上就是在setter方法中，改变依赖变量，依赖变量变化后自动触发计算属性的值

- **if-else**

```
<h1 v-if="ok">Yes</h1>
<h1 v-else>No</h1>
```

`v-else` 元素必须紧跟在带 `v-if` 或者 `v-else-if` 的元素的后面，否则它将不会被识别。

- 元素复用

  用 `key` 管理可复用的元素

- `v-if` vs `v-show`
  - 带有 `v-show` 的元素始终会被渲染并保留在 DOM 中。`v-show` 只是简单地切换元素的 CSS 属性 `display`。
  - `v-show` 不支持 `<template>` 元素
  - `v-if` 有更高的切换开销，而 `v-show` 有更高的初始渲染开销
- 当 `v-if` 与 `v-for` 一起使用时，`v-for` 具有比 `v-if` 更高的优先级。（`v-for` with `v-if`）
- **v-for与子组件**

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

- **事件**
  - 通过 `$on(eventName, eventHandler)` 侦听一个事件
  - 通过 `$once(eventName, eventHandler)` 一次性侦听一个事件
  - 通过 `$off(eventName, eventHandler)` 停止侦听一个事件
- **VNode** - Virtual Node
- vm.**$el**

  可以通过 `vm.$el` 获取dom实例；

3. 组件

3.1 注意点

- **一个组件的 `data` 必须是一个函数
  - 这样，每个实例可以维护一份被返回对象的独立的拷贝

3.2 使用方法

3.2.1 注册

全局：

```
Vue.component('my-component-name', {
  // ... options ...
})
```

局部：

```
components: {
    'my-component': () => import('./my-async-component')
  }
```

3.2.1.1 组件命名

- 两种形式：
  - **kebab-case** - 连字符

    例：my-component-name
  - **PascalCase** - 驼峰

    例：MyComponentName
- 通用性
  - 组件定义名称为驼峰式的时候，在模板（template）代码中，可以使用 `驼峰式`，也可以使用 `连字符式`，并且驼峰式首字母大写可以变小写；

    如：定义

    ```
    components: {
        'MyComponent': () => import('./my-async-component')
      }
    ```

    template代码块中可以使用的形式：

    ```
    <!-- 可以使用的形式 -->
    <MyComponent></MyComponent>
    <myComponent></myComponent>
    <My-Component></My-Component>
    <my-Component></my-Component>
    <my-component></my-component>
    ```
  - 组件定义名称为连字符式的时候，在模板（template）代码中，只可以使用 `连字符式`；

3.2.2 数据传递（子→父）

子：

```
<!-- 发送事件$emit -->
<button v-on:click="$emit('enlarge-text', 0.1)">
  Enlarge text
</button>
```

父：

```
<blog-post
  ...
  v-on:enlarge-text="postFontSize += $event"
></blog-post>
```

或者使用方法处理事件：

```
onEnlargeText: function (enlargeAmount) {
    this.postFontSize += enlargeAmount
  }
```

3.2.3 在组件上使用 `v-model`

3.2.4 插槽slot

3.2.4.1 具名slot

```
<!-- base-layout组件 -->
<div class="container">
  <header>
    <slot name="header"></slot>
  </header>

  <!-- 默认slot -->
  <main>
    <slot></slot>
  </main>

  <footer>
    <slot name="footer"></slot>
  </footer>
</div>
```

```
<!-- 在父组件中使用base-layout -->
<base-layout>
  <template slot="header">
    <h1>Here might be a page title</h1>
  </template>

  <p>A paragraph for the main content.</p>
  <p>And another one.</p>

  <p slot="footer">Here's some contact info</p>
</base-layout>
```

3.2.4.2 插槽的默认内容

3.2.4.3 作用域插槽

数据：子组件已→父组件

- ES2015 解构语法

```
var o = {p: 42, q: true};
var {p, q} = o;

console.log(p); // 42
console.log(q); // true
```

- `vm.$scopedSlots` - 获作用域插槽

3.2.5 动态组件

关键词：`is`

3.2.5.1 `<keep-alive>` 组件缓存

```
<!-- 失活的组件将会被缓存！默认是不缓存的，每次都重新生成 -->
<keep-alive>
  <component v-bind:is="currentTabComponent"></component>
</keep-alive>
```

3.2.6 异步组件

3.2.6.1 处理加载状态

`since 2.3.0`

3.2.7 边界情况

3.2.7.1 $root

扩展：vuex

3.2.7.2 $parent - 访问父级组件实例

和 `$root` 类似， `$parent` 属性可以用来从一个子组件访问父组件的实例。它提供了一种机会，可以在后期随时触达父级组件，以替代将数据以 prop 的方式传入子组件的方式。

注意：这样容易引起混乱，慎用！

3.2.7.3 访问子组件实例或子元素

3.2.7.4 递归组件

3.2.7.5 组件之间的循环引用 - 构建工具（webpack等）中解决方案

3.2.7.6 inline-template

3.2.7.7 X-Templates

3.2.7.7 通过 `v-once` 创建低开销的静态组件

3.2.8 其他

> 在 2.3.0 之前的版本中，如果一个函数式组件想要接受 props，则 `props` 选项是必须的。在 2.3.0 或以上的版本中，你可以省略 `props` 选项，所有组件上的属性都会被自动解析为 props。

4. 动画

4.1 基础

```
<transition name="fade">
    <p v-if="show">hello</p>
</transition>
```

可通过修改对应class的css，修改动画表现：

```
.fade-enter-active, .fade-leave-active {
  transition: opacity .5s;
}
.fade-enter, .fade-leave-to /* .fade-leave-active below version 2.1.8 */ {
  opacity: 0;
}
```

4.1.1 过渡的类名

4.1.2 JavaScript 钩子

> 当只用 JavaScript 过渡的时候，**在 enter 和 leave 中必须使用 done 进行回调**。否则，它们将被同步调用，过渡会立即完成。

> 推荐对于仅使用 JavaScript 过渡的元素添加 `v-bind:css="false"`，Vue 会跳过 CSS 的检测。这也可以避免过渡过程中 CSS 的影响。

4.1.3 初始渲染的过渡

```
<transition appear>
  <!-- ... -->
</transition>
```

4.1.4 过渡模式

```
<transition name="fade" mode="out-in">
  <!-- ... the buttons ... -->
</transition>
```

4.1.5 列表过渡 - 神奇

关键字：**v-move**、**<transition-group>**

4.1.6 动态状态过渡

4.2 动画插件

`TweenLite`、`lodash`、`SVG`

5. 混入

5.1 基础

```
// 定义一个混入对象
var myMixin = {
  created: function () {
    this.hello()
  },
  methods: {
    hello: function () {
      console.log('hello from mixin!')
    }
  }
}

// 定义一个使用混入对象的组件
var Component = Vue.extend({
  mixins: [myMixin]
})

var component = new Component() // => "hello from mixin!"
```

组件数据优先：

```
var mixin = {
  data: function () {
    return {
      message: 'hello',
      foo: 'abc'
    }
  }
}

new Vue({
  mixins: [mixin],
  data: function () {
    return {
      message: 'goodbye',
      bar: 'def'
    }
  },
  created: function () {
    console.log(this.$data)
    // => { message: "goodbye", foo: "abc", bar: "def" }
  }
})
```

同名钩子函数将混合为一个数组

```
var mixin = {
  created: function () {
    console.log('混入对象的钩子被调用')
  }
}

new Vue({
  mixins: [mixin],
  created: function () {
    console.log('组件钩子被调用')
  }
})

// => "混入对象的钩子被调用"
// => "组件钩子被调用"
```

值为对象的选项，例如 `methods`，`components` 和 `directives`，将被混合为同一个对象。两个对象键名冲突时，取组件对象的键值对

```
var mixin = {
  methods: {
    foo: function () {
      console.log('foo')
    },
    conflicting: function () {
      console.log('from mixin')
    }
  }
}

var vm = new Vue({
  mixins: [mixin],
  methods: {
    bar: function () {
      console.log('bar')
    },
    conflicting: function () {
      console.log('from self')
```

```
    }
  }
})

vm.foo() // => "foo"
vm.bar() // => "bar"
vm.conflicting() // => "from self"
```

全局混入，影响所有vue实例：

```
// 为自定义的选项 'myOption' 注入一个处理器。
Vue.mixin({
  created: function () {
    var myOption = this.$options.myOption
    if (myOption) {
      console.log(myOption)
    }
  }
})

new Vue({
  myOption: 'hello!'
})
// => "hello!"
```

5.2 自定义选项合并策略

可以向 `Vue.config.optionMergeStrategies` 添加一个函数：

```
// 合并策略选项分别接收在父实例和子实例上定义的该选项的值作为第一个和第二个参数，Vue 实例上下文被作为第三个参数传入。
Vue.config.optionMergeStrategies.myOption = function (toVal, fromVal, vm) {
  // return mergedVal
}
```

6. 自定义指令

6.1 全局

```
// 注册一个全局自定义指令 `v-focus`
Vue.directive('focus', {
  // 当被绑定的元素插入到 DOM 中时……
  inserted: function (el) {
    // 聚焦元素
    el.focus()
  }
})
```

6.2 局部

```
directives: {
  focus: {
    // 指令的定义
    inserted: function (el) {
      el.focus()
    }
  }
}
```

6.3 钩子函数

6.3.1 对象字面量 - 接收参数

6.3.2 函数简写

原博文中没有介绍

6.3.3 Vue.directive

```
// 注册
Vue.directive('my-directive', {
  bind: function () {},
  inserted: function () {},
  update: function () {},
  componentUpdated: function () {},
  unbind: function () {}
})

// 注册 (指令函数)
Vue.directive('my-directive', function () {
  // 这里将会被 `bind` 和 `update` 调用
})

// getter，返回已注册的指令
var myDirective = Vue.directive('my-directive')
```

7. 渲染函数

> Vue 的模板实际是编译成了 render 函数

```
Vue.component('anchored-heading', {
  render: function (createElement) {
    return createElement(
      'h' + this.level,   // tag name 标签名称
      this.$slots.default // 子组件中的阵列
    )
  },
  props: {
    level: {
      type: Number,
      required: true
    }
  }
})
```

结果类似：

```
<script type="text/x-template" id="anchored-heading-template">
  <h1 v-if="level === 1">
    <slot></slot>
  </h1>
  <h2 v-else-if="level === 2">
    <slot></slot>
  </h2>
  <h3 v-else-if="level === 3">
    <slot></slot>
  </h3>
```

```
  <h4 v-else-if="level === 4">
    <slot></slot>
  </h4>
  <h5 v-else-if="level === 5">
    <slot></slot>
  </h5>
  <h6 v-else-if="level === 6">
    <slot></slot>
  </h6>
</script>
```

```
Vue.component('anchored-heading', {
  template: '#anchored-heading-template',
  props: {
    level: {
      type: Number,
      required: true
    }
  }
})
```

7.1 `createElement` 参数

7.2 事件 & 按键修饰符 -有趣

7.3 JSX - 这是啥？

7.4 Vue.compile(template)

8. 函数式组件

```
<template functional>
</template>
```

```
  Vue.component('my-component', {
    functional: true,
    ...
  }
```

functional：使组件无状态 (没有 `data`) 和无实例 (没有 `this` 上下文)。他们用一个简单的 `render` 函数返回虚拟节点使他们更容易渲染

9. 插件

```
// 调用 `MyPlugin.install(Vue)`
Vue.use(MyPlugin)
// 或者
Vue.use(MyPlugin, { someOption: true })
// Vue.use 会自动阻止多次注册相同插件，届时只会注册一次该插件
```

> Vue.js 官方提供的一些插件 (例如 `vue-router`) 在检测到 `Vue` 是可访问的全局变量时会自动调用 `Vue.use()`。然而在例如 CommonJS 的模块环境中，你应该始终显式地调用 `Vue.use()`

当 install 方法被同一个插件多次调用，插件将只会被安装一次。

9.1 插件库

awesome-vue 集合了来自社区贡献的数以千计的插件和库

10. 过滤器

```
<!-- 在双花括号中 -->
{{ message | capitalize }}

<!-- 在 `v-bind` 中 -->
<div v-bind:id="rawId | formatId"></div>
```

```
// 注册
Vue.filter('my-filter', function (value) {
  // 返回处理后的值
})

// getter，返回已注册的过滤器
var myFilter = Vue.filter('my-filter')
```

10.1 过滤器链

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

10.2 传递参数

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

`filterA` 被定义为接收三个参数的过滤器函数。其中 `message` 的值作为第一个参数，普通字符串 `'arg1'` 作为第二个参数，表达式 `arg2` 的值作为第三个参数

11. 构建

11.1 使用构建工具

> 当使用 webpack 或 Browserify 类似的构建工具时，Vue 源码会根据 `process.env.NODE_ENV` 决定是否启用生产环境模式，默认情况为开发环境模式

11.1.1 提取组件的 CSS

11.1.2 运行时错误跟踪

11.2 整合第三方路由

12 优化

- performance

设置为 `true` 以在浏览器开发工具的性能/时间线面板中启用对组件初始化、编译、渲染和打补丁的性能追踪，查看vue性能表现；

13. 踩坑

13.1 关于 `App.data`

- 数据相互影响

```
// 我们的数据对象
    var data = { a: 1 }

    // 该对象被加入到一个 Vue 实例中
    var vm = new Vue({
      data: data
    })

    // 获得这个实例上的属性
    // 返回源数据中对应的字段
    vm.a == data.a // => true

    // 设置属性也会影响到原始数据
    vm.a = 2
    data.a // => 2

    // ……反之亦然
    data.a = 3
    vm.a // => 3
```

- > 只有当实例被创建时 `data` 中存在的属性才是**响应式**的，对于新的属性（如：`vm.b = 'hi'`），改动将不会触发任何视图的更新

- Object.**freeze**(dataObj) - 阻止修改现有的属性，也意味着响应系统无法再追踪变化

```
var dataObj = {message: 'Hello Vue!'}

Object.freeze(dataObj)
// 此时，app.message和dataObj.message变为只读，
// 但是尝试修改app.message时，会报错（因为它是只读），
// 尝试修改dataObj.message虽然不会报错，但是dataObj.message的值并没变化；

// 此时还可通过app.otherProp = value 来对app进行操作，并能获取到app.otherProp的值，
// 但是，对于dataObj，虽然可以进行dataObj.otherProp = value赋值操作，虽不报错，但实际上却是赋值失败；
var app = new Vue({
    el: '#app',
    data: dataObj
})
```

- `_` 和 `$` 开头的key

  以 `_` 或 `$` 开头的属性 **不会** 被 Vue 实例代理（不能通过vm._xxx/vm.$xxx访问），因为它们可能和 Vue 内置的属性、API 方法冲突。你可以使用例如 `vm.$data._property` 的方式访问这些属性。

- 关于this

  > 不要在选项属性或回调上使用箭头函数，比如 `created: () => console.log(this.a)` 或 `vm.$watch('a', newValue => this.myMethod())` 。因为箭头函数是和父级上下文绑定在一起的，`this` 不会是如你所预期的 Vue 实例，经常导致 `Uncaught TypeError: Cannot read property of undefined`

> 或 `Uncaught TypeError: this.myMethod is not a function` 之类的错误。

- 监视器

```
var app = new Vue({
        el: '#app',
        data: {
            message: 'Hello Vue!',
            watchProp: 'a'
        },
        watch: {
            // 监听message变化
            message: function (newMessage, oldMessage) {
                console.log(newMessage + ' : ' + oldMessage)
                this.watchProp = newMessage
            }
        }
    })
```

- **v-bind:class**
  - `v-bind:class` 接收一个**对象**

    ```
    <div v-bind:class="{ className: showBoolean }"></div>
    ```

    - className是否起作用，取决于showBoolean
    - 结合计算属性，更加美妙
  - `v-bind:class` 接收一个**数组**
    - 变体

      ```
      <div v-bind:class="[isActive ? activeClass : '', errorClass]"></div>
      ```

- **自定义组件**上使用 `class` 属性

  class将追加在根元素原class数组上，上面的**v-bind:class**情况也适用；

- **v-bind:style**
  - 接收对象，CSS 属性名可以用驼峰式 (camelCase) 或短横线分隔 (kebab-case，记得用单引号括起来) 如：

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

    ```
    data: {
      fontSize: 30
    }
    ```

  - 接收对象数组

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

  - 自动添加前缀

    没举例子，还没明白；

  - **多浏览器支持**

    从 2.3.0 起你可以为 `style` 绑定中的属性提供一个包含**多个值的数组**，常用于提供多个带前缀的值，例如：

```
<div :style="{ display: ['-webkit-box', '-ms-flexbox', 'flex'] }"></div>
```

这样写只会渲染数组中最后一个被浏览器支持的值。在本例中，如果浏览器支持不带浏览器前缀的 flexbox，那么就只会渲染 `display: flex`。

- **数组**

  由于 JavaScript 的限制，Vue 不能检测以下变动的数组：

  1. 当你利用索引直接设置一个项时，例如：`vm.items[indexOfItem] = newValue`
  2. 当你修改数组的长度时，例如：`vm.items.length = newLength`

- data修改检测

- `v-model` 会忽略所有表单元素的 `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

- 对于需要使用输入法 (如中文、日文、韩文等) 的语言，你会发现 `v-model` 不会在输入法组合文字过程中得到更新。如果你也想处理这个过程，请使用 `input` 事件

- 如果 `v-model` 表达式的初始值未能匹配任何选项，`<select>` 元素将被渲染为“未选中”状态。在 iOS 中，这会使用户无法选择第一个选项。因为这样的情况下，iOS 不会触发 change 事件。因此，更推荐像上面这样提供一个值为空的禁用选项。

```
new Vue({
  el: '...',
  data: {
    selected: ''  // 提供一个空值
  }
})
```

- `data` 选项是特例，在 `Vue.extend()` 中它必须是函数

- Vue.extend(options) - 使用基础 Vue 构造器，创建一个“子类”，类似java的继承

```
// 创建构造器
var Profile = Vue.extend({
  template: '<p>{{firstName}} {{lastName}} aka {{alias}}</p>',
  data: function () {
    return {
      firstName: 'Walter',
      lastName: 'White',
      alias: 'Heisenberg'
    }
  }
})
// 创建 Profile 实例，并挂载到一个元素上。
new Profile().$mount('#mount-point')
```

- Vue.nextTick()

> Vue是异步执行dom更新的，一旦观察到数据变化，Vue就会开启一个队列，然后把在同一个事件循环 (event loop) 当中观察到数据变化的 watcher 推送进这个队列。如果这个watcher被触发多次，只会被推送到队列一次。这种缓冲行为可以有效的去掉重复数据造成的不必要的计算和DOm操作。而在下一个事件循环时，Vue会清空队列，并进行必要的DOM更新。
>
> 当你设置 vm.someData = 'new value'，DOM 并不会马上更新，而是在异步队列被清除，也就是下一个事件循环开始时执行更新时才会进行必要的DOM更新。如果此时你想要根据更新的 DOM 状态去做某些事情，就会出现问题。。为了在数据变化之后等待 Vue 完成更新 DOM，可以在数据变化之后立即使用 Vue.nextTick(callback) 。这样回调函数在 DOM 更新完成后就会调用。

- ## Vue.set(target, key, value)

  此函数添加新属性是**响应式**的

- Vue.delete(target, key)

  删除属性 - 类似 `Vue.set`，也是**响应式**的

- Vue.version

- 深拷贝

  通过将 `vm.$data` 传入 `JSON.parse(JSON.stringify(...))` 得到深拷贝的原始数据对象

- 当一个**组件**被定义，`data` 必须声明为返回一个初始数据对象的函数，因为组件可能被用来创建多个实例。如果 `data` 仍然是一个纯粹的对象，则所有的实例将**共享引用**同一个数据对象！

- props 可以是数组或对象，用于接收来自父组件的数据。props 可以是简单的数组，或者使用对象作为替代，对象允许配置高级选项，如类型检测、自定义校验和设置默认值

```
// 数组语法，略

// 对象语法，提供校验
Vue.component('props-demo-advanced', {
  props: {
    // 检测类型
    height: Number,
    // 检测类型 + 其他验证
    age: {
      type: Number,
      default: 0,
      required: true,
      validator: function (value) {
        return value >= 0
      }
    }
  }
})
```

- propsData

  主要用于测试：

```
var Comp = Vue.extend({
  props: ['msg'],
  template: '<div>{{ msg }}</div>'
})

var vm = new Comp({
  propsData: {
    msg: 'hello'
  }
})
```

- 关于computed

  如果你为一个计算属性使用了箭头函数，则 `this` 不会指向这个组件的实例，不过你仍然可以将其实例作为函数的第一个参数来访问：

```
computed: {
  aDouble: vm => vm.a * 2
```

```
}
```

- **watch**进阶

```
var vm = new Vue({
  data: {
    a: 1,
    b: 2,
    c: 3,
    d: 4,
    e: {
      f: {
        g: 5
      }
    }
  },
  watch: {
    a: function (val, oldVal) {
      console.log('new: %s, old: %s', val, oldVal)
    },
    // 方法名
    b: 'someMethod',
    // 深度 watcher
    c: {
      handler: function (val, oldVal) { /* ... */ },
      deep: true
    },
    // 该回调将会在侦听开始之后被立即调用
    d: {
      handler: function (val, oldVal) { /* ... */ },
      immediate: true
    },
    e: [
      function handle1 (val, oldVal) { /* ... */ },
      function handle2 (val, oldVal) { /* ... */ }
    ],
    // watch vm.e.f's value: {g: 5}
    'e.f': function (val, oldVal) { /* ... */ }
  }
})
vm.a = 2 // => new: 2, old: 1
```

- 提供的元素（** **）只能作为挂载点。不同于 Vue 1.x，所有的挂载元素会被 Vue 生成的 DOM 替换。因此不推荐挂载 root 实例到 `<html>` 或者 `<body>` 上。
- provide / inject
- 件在全局用 `Vue.component()` 注册时，全局 ID 自动作为组件的 name
- 有名字（name）的组件有更友好的警告信息
- 双括号（`{{varName}}`）可以更改 - delimiters
- 自定义v-model关联属性/事件

  v-model默认关联value属性和input事件：

```
<custom-component v-model="custom" />
<!-- 等效 -->
<custom-component :value="custom" @input="val => custom = val" />
```

  自定义：

```
export default {
  name: 'custom-component',
  model: {
    prop: 'customVal',
    event: 'customEvent'
  }
}
```

```
<custom-component :customVal="custom" @customEvent="val => custom = val" />
```

- **inheritAttrs**

  用于父子组件之间的数据传递：

  博客：vm.$attrs与详解 Vue 2.4.0 带来的 4 个重大变化

- **commonts**

  编译后，是否保留HTML注释；

- vm.$options
- vm.$root
- vm.$slots
- vm.$isServer
- vm.$forceUpdate() - 强制刷新组件
-

14. 相关网站

- Vue.js
- API