

- 课程准备
- 开始使用Vue
 - `el`
 - `$mount`
 - `data`
 - 插值表达式
- vue的响应式-1
 - `vm.$el`
 - `vm.$nextTick & Vue.nextTick`
- vue的响应式-2
- 扩展_剖析Vue响应式原理
- Vue相关指令
 - `v-pre`
 - `v-cloak`
 - `v-once`
 - `v-text`
 - `v-html`
- 条件渲染
 - `v-if`
 - `v-else`
 - `v-else-if`
 - `v-show`
 - `v-if VS v-show`
- `v-bind` 指令
- `v-on`指令
 - 为什么在 HTML 中监听事件?
- `v-on`指令的修饰符
 - 事件修饰符
 - `.stop`
 - `.prevent`
 - `.capture`
 - `.self`
 - `.once`
 - `.passive`
 - 注意
 - 按键修饰符
 - 按键码
 - 系统修饰键
 - `exact` 修饰符
 - 鼠标按钮修饰符
- 列表渲染
 - 在`v-for`中使用数组
 - 在`v-for`中使用对象
 - 在`v-for`中使用数字
 - 在`v-for`中使用字符串
 - 循环一段包含多个元素的内容
 - 关于`key`
 - `key`的使用方法
 - `v-for` 和 `v-if` 一同使用
- 练习_仿淘宝商品筛选
- 练习_todoList
- `v-model`指令
 - `input`
 - `type=text` 文本框
 - `type=checkbox` 复选框
 - 单个复选框
 - 多个复选框
 - `type=radio` 单选框

- `textarea`
 - `select`
 - 单选
 - 多选
 - 修饰符
 - `.lazy`
 - `.number`
 - `.trim`
 - 练习_简易计算器
 - 练习_调查问卷
 - 计算属性
 - 基础用法
 - 计算属性 vs 方法
 - 深入计算属性
 - `get` 读取
 - `set` 设置
 - 练习_姓名筛选
 - 练习_全选商品
 - 侦听器
 - 值类型
 - 函数类型
 - 字符串类型
 - 对象类型
 - `handler`
 - `deep`
 - `immediate`
 - 数组类型
 - 键类型
 - 正常对象key值
 - 字符串类型key值
 - `vm.$watch`
 - 侦听器 vs 计算属性
- 练习_仿百度搜索联想
- vue-resource
 - 引入vue-resource
 - 请求方法
 - POST请求
 - GET请求
 - PUT请求
 - PATCH请求
 - DELETE请求
 - HEAD请求
 - JSONP请求
 - options 参数说明
 - 响应对象
 - 属性
 - 方法
 - 最后的话
- Axios
 - 引入
 - API
 - `config` 配置对象
 - 方法别名
 - 配置默认值
 - 全局配置
 - 实例配置
 - 请求配置
 - 配置的优先顺序

- 并发
- 拦截器
 - 请求拦截器
 - 响应拦截器
 - 移除拦截器
 - 为axios实例添加拦截器
- 取消请求
- 错误处理
- axios 预检
- template 选项
- Vue生命周期
 - 生命周期图示
 - 生命周期钩子
 - beforeCreate
 - created
 - beforeMount
 - mounted
 - beforeUpdate
 - updated
 - beforeDestroy
 - destroyed
- 练习_bilibili首页
 - 组件是什么?
 - 组件注册
 - 全局组件
 - 局部组件
 - 组件名
 - 组件复用
 - 自闭合组件
 - 组件的data选项
 - 单个根元素
- 组件_Prop
 - 注册自定义特性
 - Prop的大小写
 - 传递静态或动态 Prop
 - 传递一个对象的所有属性
- 组件_Prop验证
- 组件_单向数据流
- 组件_非Prop特性
 - 替换/合并已有的特性
 - 禁用特性继承
- 组件_监听组件事件
 - 使用事件抛出一个值
 - 事件名
 - 将原生事件绑定到组件
 - 在组件上使用 v-model
 - .sync 修饰符
 - v-model VS .sync
- 组件_插槽
 - 插槽内容
 - 编译作用域
 - 后备内容
 - 具名插槽
 - 作用域插槽
 - 独占默认插槽的缩写语法（不能跟具名插槽同时使用）
 - 解构插槽Prop
 - 动态插槽名
 - 具名插槽的缩写

- 废弃了的语法
 - 带有slot特性的具名插槽
 - 带有slot-scope特性的作用域插槽
- 组件_动态组件
 - 基本使用
 - keep-alive
 - activated & deactivated
- 组件_处理边界情况
 - 访问元素 & 组件
 - 访问根实例
 - 访问父级组件实例
 - 依赖注入
 - 访问子组件实例或子元素
 - 程序化的事件侦听器
 - 循环引用
 - 递归组件
 - 组件之间的循环引用
 - 模板定义的替代品
 - 内联模板
 - X-Template
 - 控制更新
 - 强制更新
 - 通过v-once创建低开销的静态组件
- 组件_通信
 - props (推荐)
 - \$emit (推荐)
 - v-model 双向绑定
 - .sync 实现双向绑定
 - \$attrs 不推荐
 - \$listeners
 - \$root
 - \$parent
 - \$children
 - ref
 - provide & inject
 - eventBus(事件总线) 兄弟组件中传输数据 不推荐
 - Vuex
- 混入
 - 基础
 - 选项合并
 - 全局混入(谨慎使用)
- 自定义指令
 - 简介
 - 钩子函数
 - bind
 - inserted
 - update
 - componentUpdated
 - unbind
 - 钩子函数参数
 - 练习
 - 模拟 v-show
 - 模拟 v-model
 - 写一个 v-slice (截取文本框)
 - 动态指令参数
 - 函数简写
 - 对象字面量
- 过滤器

- 定义过滤器
- 参数
- 过滤器串联
- 练习
 - 首字母大写
 - 数字中间加上逗号
 - 数字添加文字“万”
- 安装脚手架
 - 安装@vue/cli
 - 快速原型开发
 - 组件结构
 - 安装vscode插件
- 练习_树形组件
- 利用脚手架搭建项目
- 渲染函数
 - 基础
 - 节点、树、以及虚拟DOM
 - 虚拟DOM
 - createElement参数
 - 深入 ‘与模板中属性对应的数据对象’
 - 使用JavaScript代替模板功能
 - v-if 和 v-for
 - v-model
 - 事件&按键修饰符
 - 插槽
- JSX (JS + XML(html) == JSX)
 - 插值
 - 指令
 - v-text
 - v-html
 - v-show(控制是否显示)
 - v-if
 - v-for
 - v-on
 - v-bind
 - v-model
 - v-pre
 - v-cloak
 - v-once
 - Ref
 - 自定义指令 (没有尝试, 对错未定)
 - 过滤器
 - 插槽
- 函数式组件
 - slots() VS children
 - 基于模板的函数式组件
- 过渡_单元素过渡
 - 单元素/组件的过渡
 - 过渡的类名
 - 类名前缀
 - CSS 动画
 - 自定义过渡的类名
 - 同时使用过渡和动画
 - 显性的过渡时间
 - 初始渲染的过渡
 - JavaScript 钩子
- 过渡_多元素过渡 (切换展示多元素)
 - 过渡模式

- 多组件过渡
- 过渡_列表过渡
 - 列表的排序过渡
 - 列表的交错过渡
- 过渡_复用过渡
- 组件_异步组件
- **VueRouter_基础**
 - 什么是路由?
 - 什么时候使用前端路由?
 - 安装路由
 - 使用路由
 - `JavaScript`
 - `html`
 - `router-link class`
 - hash模式
 - history 模式
- **VueRouter_命名路由-嵌套路由-重定向-别名**
 - 命名路由
 - 嵌套路由
 - 重定向
 - 别名
- **VueRouter_编程式的导航**
 - `$router`
 - `$router.push`
 - `$router.replace`
 - `$router.go(n)`
 - `$route`
 - `$route.path`
 - `$route.params`
 - `$route.query`
 - `$route.hash`
 - `$route.fullPath`
 - `$route.matched`
 - `$route.name`
 - `$route.redirectedFrom`
- **VueRouter_动态路由匹配**
- **VueRouter_命名视图-路由组件传参**
 - 命名视图
 - 路由组件传参
 - 布尔模式
 - 对象模式
 - 函数模式
- **VueRouter_导航守卫**
 - 全局守卫
 - 全局前置守卫 `beforeEach`
 - 全局解析守卫 `beforeResolve`
 - 全局后置钩子 `afterEach`
 - 路由独享守卫
 - `beforeEnter`
 - 组件内守卫 (页面级守卫)
 - `beforeRouteEnter`
 - `beforeRouteUpdate`
 - `beforeRouteLeave`
 - 完整的导航解析流程
- **VueRouter_路由元信息**
- **VueRouter_过渡动效-滚动行为**
 - 过渡动效
 - 滚动行为

- **Vuex_State**
 - 安装
 - 使用
 - **State**
 - 在Vue组件中获得Vuex状态
 - `mapState` 辅助函数
- **Vuex_Getter**
 - 通过属性访问
 - 通过方法访问
 - `mapGetters` 辅助函数
- **Vuex_Mutation**
 - 在组件中提交 `Mutation`
 - 提交载荷 (`Payload`)
 - 对象风格的提交方式
 - 使用常量替代 `Mutation` 事件类型
 - `Mutation` 需遵守 Vue 的响应规则
 - 表单处理
 - 双向绑定的计算属性
 - `Mutation` 必须是同步函数
 - 严格模式
 - 开发环境与发布环境
- **Vuex_Action**
 - 分发Action
 - 组合 Action
 - Vuex 管理模式
- **Vuex_Module(模块)**
 - 命名空间
 - 模块的局部状态

课程准备

- 所有代码及文件放到github上进行托管，同学们自行clone
- github地址: <https://github.com/DuYi-Edu/DuYi-VUE>
- 下载并安装vue-devtools插件
- 由于某些原因，下载不了谷歌商店上的插件，所以做好科学上网
- 安装插件: <https://github.com/qin-ziqi/Chrome-SetupVPN-3.7.0>，跟着上面的步骤做，插件安装较慢，据不完全统计，约需30分钟
- 安装好了后，点击谷歌浏览器的更多工具 -> 扩展程序 -> 打开chrome网上应用商店 -> 搜索vue devtools -> 添加至chrome
- 是否生效？访问: <https://www.bilibili.com/>，图标变绿了就生效了
- 有什么用？以后再说，听我的，安装
- 课程笔记会给你们，所以在学习课程的时候，认真听，认真思考
- 安装vscode 插件: Markdown Preview Enhanced
- 课程会先讲解vue语法，先把vue语法学会了之后，在剖析其他的问题

开始使用Vue

1. 引入vue.js
 - | 官网: vuejs.org
 - | 开发版本：包含完整的警告和调试模式
 - | 生产版本：删除了警告，体积更小
2. 引入vue.js后，给我们提供了一个构造函数 `Vue`
3. 在js中，`new Vue()`
4. `new Vue()` 后会返回一个vue实例对象，我们用变量接着它
5. `const vm = new Vue()`
6. 传递一个配置对象{} --> `const vm = new Vue({})`

el

类型: 字符串

全称: element (元素)

作用: 配置控制的元素, 表示Vue要控制的区域, 值为css选择器

```
<!-- 被Vue控制的区域, 我们称之为模板 -->
<div id="app"></div>
```

```
const vm = new Vue({
  el: '#app' // 控制id为app的元素
})
```

\$mount

- 作用和el一致, 都是配置控制的元素, 使用哪个都可以, 二选一

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

```
const vm = new Vue({})
vm.$mount('#app');
```

- 问: 和el有什么不同?

答: 本质上没什么不同, \$mount为手动挂载, 在项目中有时要进行延迟挂载, 比如有时要在挂载之前进行一些其他的操作, 比如判断等等 (但是, 这样做的时候很少, 比邓哥回家的次数还少, emmmmm)

data

- 类型: 对象

- 作用: 存放要用到的数据, 数据为响应式的

```
const vm = new Vue({
  el: '#app',
  data: {
    'mrDeng': '风姿绰约、花枝招展'
  }
})
```

插值表达式

- 使用方法: {{ }}

- 可以将vue中的数据填在插值表达式中, 如:

```
<div id="app">{{ mrDeng }}</div>
```

```
const vm = new Vue({
  el: '#app',
  data: {
    mrDeng: '邓哥: 风姿绰约、花枝招展'
  }
})
```

- 除了填写data之外, 还可以直接填写数据值 (数字、字符串、布尔值、undefined、null、数组、对象), 如:

```
<div id="app">
  {{ 5201314 }}
  {{ '婀娜多姿、亭亭玉立' }}
  {{ true }}
  {{ ['邓旭明', '小刘', '王小宝'] }}
  {{ {name: '邓旭明', age: 80, height: '140cm', weight: '100kg'} }}
</div>
```

- 注意：在插值表达式中直接书写对象类型值时，不要将三个{}连在一起，这样会报错，如：

```
<div id="app">
  <!-- 这样可是不行滴 -->
  {{{{name: '邓旭明', age: 80, height: '140cm', weight: '100kg'}}}}
</div>
```

- 还可在插值表达式中写表达式，如：

```
<div id="app">
  <!-- 运算表达式 -->
  {{ 'you' + 'me' }}
  {{ 10 - 5 }}
  {{ 100 * 7 }}
  {{ 1000 / 12 }}
  <!-- 逻辑表达式 -->
  {{ liu || li }}
  {{ deng && liu }}
  {{ !wang }}
  <!-- 三元表达式 -->
  {{ 1 + 1 === 3 ? '邓旭明' : '正常人' }}
  <!-- 函数调用也是表达式，也可以使用，这个以后再学哈... -->
</div>
```

- 还可以填写其他的吗？不可以，No，以下这些都是不行滴：

```
<div id="app">
  <!-- 这是语句，不可以写在插值表达式中 -->
  {{ var Deng = 'shuaige'; console.log(deng) }}
  <!-- 流程控制也不可以 -->
  {{ if(Deng.looks === 'shuai'){ console.log('不可能') } }}
</div>
```

- 记住**：插值表达式中，可以写：data、js数据、表达式，其他的想都不要想。
- 注意**：只要插值表达式中使用了数据，必须在data中声明过，否则会报错

```
<!-- 此时就报错啦，因为mrCheng，未在data中声明过 -->
<div id="app">
  {{ mrCheng }}
</div>
```

```
const vm = new Vue({
  el: '#app',
  data: {
    mrDeng: '邓哥：风姿绰约、花枝招展'
  }
})
```

- 还有另外一种可能，使用了未被声明过的数据，不报错：

```
<!-- 此时不报错啦，why? -->
<!-- 在作用域上找不到，报错 -->
<!-- 在原型链上找不到，值为undefined -->
<!-- undefined为js基本类型值，所以就不报错啦 -->
<div id="app">
  {{ mrDeng.wife }}
</div>
```

```
const vm = new Vue({
  el: '#app',
  data: {
    mrDeng: {
      name: '邓旭明',
      age: 80,
      height: '140cm',
      weight: '100kg'
    }
  }
})
```

vue的响应式-1

- 数据变化，页面就会重新渲染
- 怎么更改数据？so easy

```
<div id="app">
  {{ mrDeng }}
</div>

const vm = new Vue({
  el: '#app',
  data: {
    mrDeng: '邓哥：风姿绰约、花枝招展'
  }
});
vm.mrDeng = '手如柔荑、肤如凝脂' // 见证奇迹的时刻，页面变化啦
```

- 问：为什么data会直接出现在vm实例对象中咧？

答：当创建vue实例时，vue会将data中的成员代理给vue实例，目的是为了实现响应式，监控数据变化，执行某个监听函数（怎么实现的？想一想，提示：Object.defineProperty，试着实现一下）

- 问：实例中除了data数据外，其他东西是啥子？

为了防止名称冲突。因为会将data中数据代理给vue，假如说我们自己写的data名称和vue中自带的属性冲突了，那么就会覆盖vue内部的属性，所以vue会把自己内部的属性成员名称前加上\$或_，如果加上的是\$，代表是我们可以使用的，如果加上的是_，是vue自己内部使用的方法或属性，我们不需要调用

- 更改的数据必须是存在的数据，否则不能重新渲染页面，因为他监听不到，如：

```
<!-- 即使更改了数据，也不会重新渲染页面 -->
<div id="app">
  {{ mrDeng.wife }}
</div>
```

```
const vm = new Vue({
  el: '#app',
  data: {
    mrDeng: {
      name: '邓旭明',
      age: 80,
      height: '140cm',
      weight: '100kg'
    }
  }
});

vm.mrDeng.wife = 'liu';
```

- 更改的数据必须已渲染过的数据，否则从性能角度考虑，不会重新渲染页面，如：

```
<!-- 即使更改了数据，也不会重新渲染页面 -->
<div id="app">
  {{ mrDeng.wife }}
</div>
```

```

const vm = new Vue({
  el: '#app',
  data: {
    msg: '邓哥：风姿绰约、花枝招展',
    mrDeng: {
      name: '邓旭明',
      age: 80,
      height: '140cm',
      weight: '100kg'
    }
  }
})
vm.mrDeng.wife = 'liu';
vm.msg = '邓哥：手如柔荑、肤如凝脂';

```

- 更改数据后，页面会立刻重新渲染吗？

vue更新DOM的操作是异步执行的，只要侦听到数据变化，将开启一个异步队列，如果一个数据被多次变更，那么只会被推入到队列中一次，这样可以避免不必要的计算和DOM操作。

同步执行栈执行完毕后，会执行异步队列

```

<div id="app">{{ msg }}</div>

const vm = new Vue({
  el: '#app',
  data: {
    msg: '杉杉'
  }
})
vm.msg = '杉杉超美的';
console.log(vm.msg); // 杉杉超美的，此时数据已更改
console.log(vm.$el.innerHTML); // 杉杉。此时页面还未重新渲染

```

vm.\$el

- 值为被Vue控制的元素（或者说，Vue挂载的元素）

vm.\$nextTick & Vue.nextTick

- 如何在更改数据后，看到渲染后的页面上的值？

答：利用`vm.nextTick`或`Vue.nextTick`，在页面重新渲染，*DOM*更新后，会立刻执行`vm.nextTick`

```

<div id="app">{{ msg }}</div>

const vm = new Vue({
  el: '#app',
  data: {
    msg: '杉杉'
  }
})
vm.msg = '杉杉超美的';
console.log(vm.msg); // 杉杉超美的，此时数据已更改
// 1. 使用vm.$nextTick
vm.$nextTick(() => {
  console.log(vm.$el.innerHTML); // 杉杉超美的
})
// 2. 使用Vue.nextTick
Vue.nextTick(() => {
  console.log(vm.$el.innerHTML); // 杉杉超美的
})

```

- `vm.nextTick`和`Vue.nextTick`还可以作为Promise使用

```

<div id="app">{{ msg }}</div>

const vm = new Vue({
  el: '#app',
  data: {
    msg: '杉杉'
  }
})
vm.msg = '杉杉超美的';
// 1. 使用vm.$nextTick
vm.$nextTick().then(() => {
  console.log(vm.$el.innerHTML); // 杉杉超美的
})
// 2. 使用Vue.nextTick
Vue.nextTick().then(() => {
  console.log(vm.$el.innerHTML); // 杉杉超美的
})

```

- `vm.$nextTick` 和 `Vue.nextTick`的区别?

||| `Vue.nextTick`内部函数的`this`指向`window`

```

Vue.nextTick(function () {
  console.log(this); // window
})

```

||| `vm.$nextTick`内部函数的`this`指向`Vue`实例对象

```

vm.$nextTick(function () {
  console.log(this); // vm实例
})

```

- 好奇`nextTick`是怎么实现的吗?
- 异步任务分为宏任务 (macro) 和微任务 (micro)
- 宏任务比较慢 (如`setTimeout`等), 微任务比较快 (如`Promise.then()`等)
- 微任务在前, 宏任务在后 (eventloop, 事件环)

```

// 控制台打印顺序: promise > timeout
setTimeout(() => {
  console.log('timeout');
}, 0)
Promise.resolve().then(() => {
  console.log('promise');
})

```

- 在`nextTick`的实现源码中, 会先判断是否支持微任务, 不支持后, 才会执行宏任务

```

if(typeof Promise !== 'undefined') {
  // 微任务
  // 首先看一下浏览器中有没有promise
  // 因为IE浏览器中不能执行Promise
  const p = Promise.resolve();

} else if(typeof MutationObserver !== 'undefined') {
  // 微任务
  // 突变观察
  // 监听文档中文字的变化, 如果文字有变化, 就会执行回调
  // vue的具体做法是: 创建一个假节点, 然后让这个假节点稍微改动一下, 就会执行对应的函数
} else if(typeof setImmediate !== 'undefined') {
  // 宏任务
  // 只在IE下有
} else {
  // 宏任务
  // 如果上面都不能执行, 那么则会调用setTimeout
}

```

- 曾经`vue`用过的宏任务

- `MessageChannel` 消息通道 宏任务

vue的响应式-2

- 除了未被声明过和未被渲染的数据外，还有什么数据更改后不会渲染页面？

1. 利用索引直接设置一个数组项时：

```
<!-- 即使向数组中添加了第4项，数组仍然显示3项 -->
<!-- 咳咳，一家三口，有第4个人也不能摆出来给大家看呀~ -->
<div id="app">{{ list }}</div>
```

```
const vm = new Vue({
  el: '#app'
  data: {
    dengFamily: ['邓哥', '小刘', '王小宝']
  }
})
vm.dengFamily[3] = '铁锤妹妹'; // 不是响应式的
```

2. 修改数组的长度时：

```
<!-- 更改了数组长度后，数组仍然显示1项 -->
<div id="app">{{ list }}</div>
```

```
const vm = new Vue({
  el: '#app'
  data: {
    dengWife: ['小刘']
  }
})
vm.dengWife.length = 0; // 不是响应式的
```

3. 添加或删除对象：

```
<!-- 身高还是那个身高，媳妇也只有一个，不要痴心妄想 -->
<div id="app">{{ deng }}</div>
```

```
const vm = new Vue({
  el: '#app'
  data: {
    deng: {
      wife: '小刘',
      son: '王小宝',
      weight: '100kg',
      height: '140cm',
      age: 60
    }
  }
})
vm.deng.secondWife = '铁锤妹妹'; // 不是响应式的
delete vm.deng.height; // 不是响应式的
```

- 问：要如何响应式的更新数组和对象？

更改数组：

- 利用数组变异方法：push、pop、shift、unshift、splice、sort、reverse
- 利用vm.\$set/Vue.set实例方法
- 利用vm.\$set或Vue.set删除数组中的某一项

vm.\$set是Vue.set的别名

使用方法：Vue.set(object, propertyName, value)，也就是这个意思：Vue.set(具体数组或对象，改它的第几项或key值，改成啥值)

vm.\$delete是Vue.delete的别名

使用方法：Vue.delete(object, target)，也就是这个意思：Vue.delete(要删除谁的值，删除哪个 索引或key)

```
<!-- 从此，一家三口过上了愉快生活 -->
<div id="app">{{ list }}</div>
```

```
const vm = new Vue({
  el: '#app'
  data: {
    dengFamily: ['邓哥', '小刘', '王小宝']
  }
})
// 使用数组变异方法
vm.dengFamily.push('铁锤妹妹');
// 使用vm.$set
vm.$set(vm.dengFamily, 3, '铁锤妹妹');
```

```
<!-- 邓哥的媳妇多了起来~ -->
<div id="app">{{ list }}</div>
```

```
const vm = new Vue({
  el: '#app'
  data: {
    dengWife: ['小刘']
  }
})
// 更改长度时，可以用数组的splice方法
vm.dengWife.splice(100);
```

更改对象：

1. 添加利用vm.\$set/Vue.set实例方法
2. 删除利用vm.\$delete/Vue.delete方法

```
<div id="app">{{ deng }}</div>
```

```
const vm = new Vue({
  el: '#app'
  data: {
    deng: {
      wife: '小刘',
      son: '王小宝',
      weight: '100kg',
      height: '140cm',
      age: 60
    }
  }
})
// 添加
vm.$set(vm.deng, 'secondWife', '铁锤妹妹');
// 删除
vm.$delete(vm.deng, 'height')
```

• 总结：

更改数组用变异方法，就够了
更改对象就用vm.\$set和vm.\$delete

• 问题解决了，但是为什么会这样呢？

Object.defineProperty的锅，咱们下节课说~

扩展_剖析Vue响应式原理

```

const data = {
  name: 'shanshan',
  age: 18,
  shan: {
    name: 'shanshan',
    age: 18,
    obj: {}
  },
  arr: [1, 2, 3]
}

const arrayProto = Array.prototype;
const arrayMethods = Object.create(arrayProto);
['push', 'pop', 'shift', 'unshift', 'sort', 'splice', 'reverse'].forEach(method => {
  arrayMethods[method] = function () {
    arrayProto[method].call(this, ...arguments);
    render();
  }
})

function defineReactive (data, key, value) {
  observer(value);
  Object.defineProperty(data, key, {
    get () {
      return value;
    },
    set (newVal) {
      if(value === newVal) {
        return;
      }
      value = newVal;
      render();
    }
  })
}

function observer (data) {
  if(Array.isArray(data)) {
    data.__proto__ = arrayMethods;
    return;
  }

  if(typeof data === 'object') {
    for(let key in data) {
      defineReactive(data, key, data[key])
    }
  }
}

function render () {
  console.log('页面渲染啦');
}

function $set (data, key, value) {
  if(Array.isArray(data)) {
    data.splice(key, 1, value);
    return value;
  }
  defineReactive(data, key, value);
  render();
  return value;
}

function $delete(data, key) {
  if(Array.isArray(data)) {
    data.splice(key, 1);
    return;
  }
  delete data[key];
  render();
}

observer(data);

```

利用Object.defineProperty实现响应式的劣势

1. 天生就需要进行递归
2. 监听不到数组不存在的索引的改变
3. 监听不到数组长度的改变
4. 监听不到对象的增删

Vue相关指令

- 具有特殊含义、拥有特殊功能的特性
- 指令带有v-前缀，表示它们是Vue提供的特殊特性
- 指令可以直接使用data中的数据

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。

```
<!-- 不会被编译 -->
<span v-pre>{{ msg }}</span>
```

v-cloak

- 这个指令保持在元素上直到关联实例结束编译
- 可以解决闪烁的问题
- 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

```
[v-cloak] {
  display: none;
}
```

```
<!-- {{ message }}不会显示，直到编译结束 -->
<div v-cloak>
  {{ message }}
</div>
```

v-once

- 只渲染元素一次。随后的重新渲染，元素及其所有的子节点将被视为静态内容并跳过。这可以用于优化更新性能

```
<!-- 单个元素 -->
<span v-once>{{msg}}</span>
<!-- 有子元素 -->
<div v-once>
  <h1>comment</h1>
  <p>{{msg}}</p>
</div>
```

v-text

- 更新元素的 textContent

```
<span v-text="msg"></span>
<!-- 和下面的一样 -->
<span>{{msg}}</span>
```

v-text VS Mustache {{xxx}}

- v-text替换元素中所有的文本，Mustache只替换自己，不清空元素内容

```
<!-- 渲染为：<span>杉杉最美</span> -->
<span v-text="msg">----</span>
<!-- 渲染为：<span>----杉杉最美----</span> -->
<span>----{{msg}}----</span>
```

- v-text 优先级高于 {{ }}

textContent VS innerText

1. 设置文本替换时，两者都会把指定节点下的所有子节点也一并替换掉。
2. textContent 会获取所有元素的内容，包括 <script> 和 <style> 元素，然而 innerText 不会获取所有元素。
3. innerText 受 CSS 样式的影响，并且不会返回隐藏元素的文本，而textContent 会。
4. 由于 innerText 受 CSS 样式的影响，它会触发重排 (reflow)，但textContent 不会。
5. innerText 不是标准制定出来的 api，而是IE引入的，所以对IE支持更友好。textContent 虽然作为标准方法但是只支持IE8+以上的浏览器，在最新的浏览器中，两个都可以使用。
6. 综上，Vue这里使用textContent是从性能的角度考虑的。

测试一下innerText & textContent两者性能

```
<ul class="list">
<li>1</li>
<!-- 此处省略998个 -->
<li>1000</li>
</ul>

const oList = document.getElementById("list");

console.time("innerText");
for(let i = 0; i < oList.childElementCount; i++){
  ul.children[i].innerText="innerText";
}
console.timeEnd("innerText");

console.time("textContent");
for(let i = 0; i < oList.childElementCount; i++){
  ul.children[i].textContent="innerText";
}
console.timeEnd("textContent");
```

v-html

- 更新元素的innerHTML
- 注意：内容按普通 HTML 插入，不会作为 Vue 模板进行编译
- 在网站上动态渲染任意 HTML 是非常危险的，因为容易导致 XSS 攻击。只在可信内容上使用 v-html，永不用在用户提交的内容上。

```
<input type="text" />
<button>点击</button>
<div id="app">
  <div v-html="msg"></div>
</div>

const vm = new Vue({
  el: '#app',
  data: {
    msg: 'hello world'
  }
})

const oInput = document.getElementsByTagName('input')[0];
const oButton = document.getElementsByTagName('button')[0];
let msg = null;
oButton.onclick = function () {
  vm.msg = oInput.value;
}
```

条件渲染

v-if

- 用于条件性地渲染一块内容。这块内容只会在指令的表达式返回 `truthy` 值的时候被渲染。`truthy` 值（真值）为除了5个错值得值

切换多个元素

- 因为 `v-if` 是一个指令，所以必须将它添加到一个元素上，但是如果想切换多个元素呢？此时可以把一个 `<template>` 元素当做不可见的包裹元素，并在上面使用 `v-if`。最终的渲染结果将不包含 `<template>` 元素

```
<template v-if="ok">
  <h1>Title</h1>
  <p>Paragraph 1</p>
  <p>Paragraph 2</p>
</template>
```

v-else

- 为 `v-if` 或者 `v-else-if` 添加“`else` 块”。
- 注意：前一兄弟元素必须有 `v-if` 或 `v-else-if`

```
<div v-if="Math.random() > 0.5">
  杉杉
</div>
<div v-else>
  你看不见杉杉啦
</div>
```

v-else-if

- 表示 `v-if` 的“`else if` 块”。可以链式调用。
- 注意：前一兄弟元素必须有 `v-if` 或 `v-else-if`

```
<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>
```

v-show

- 根据表达式之真假值，切换元素的 `display` CSS 属性。

```
<h1 v-show="ok">Hello!</h1>
```

v-if VS v-show

1. `v-if` 是惰性的，如果在初始渲染时条件为假，则什么也不做，直到条件第一次变为真时，才会开始渲染条件块。`v-show`则不管初始条件是什么，元素总是会被渲染，并且只是简单地基于 CSS 进行切换。
2. `v-if` 有更高的切换开销，`v-show` 有更高的初始渲染开销，如果需要非常频繁地切换，则使用 `v-show` 较好，如果在运行时条件很少改变，则使用 `v-if` 较好
3. `v-show`不支持 `<template>` 元素
4. `v-show`不支持`v-else/v-else-if`

v-bind 指令

- 动态地绑定一个或多个特性
- `:`后的为传递的参数

```

<!-- 绑定一个属性 -->


const vm = new Vue({
  el: "#app",
  data: {
    imageSrc: '...'
  }
})

<!-- 动态特性名 (2.6.0+) -->
<div id="app" v-bind:[key]='a'>赵拜拜</div>

const vm = new Vue({
  el: "#app",
  data: {
    key: 'content',
    a: 'hello'
  }
})

<!-- 结果 -->
<div id="app" content="hello">赵拜拜</div>

<!-- 缩写 -->


const vm = new Vue({
  el:"img",
  data:{
    imageSrc:'....'
  }
})

<!-- 结果 -->


<!-- 动态特性名缩写 (2.6.0+) -->
<button :[key]="value"></button>
<!-- 假定vue data中定义 key:name value:hello -->
<button name="hello"></button>

<!-- 内联字符串拼接 注意: 想拼接字符串外需要加引号-->


```

- 没有参数时，可以绑定到一个包含键值对的对象。注意此时 `class` 和 `style` 绑定不支持数组和对象。

```

<!-- 绑定一个有属性的对象 如果属性名中有横杠, 那么属性名外必须有引号-->
<div v-bind="{ id: someProp, 'other-attr': otherProp }"></div>
<!-- 结果 -->
<div id="..." other-attr="..."></div>

```

- 由于字符串拼接麻烦且易错，所以在绑定 `class` 或 `style` 特性时，Vue做了增强，表达式的类型除了字符串之外，还可以是数组或对象。

- 绑定`class`
 - 对象语法

```
<div id="app" v-bind:class="{ red: isRed }"></div>
```

```

const vm = new Vue({
  el: "#app",
  data: {
    isRed: true
  }
})

```

```
<!-- 结果 -->
<div id="app" class="red"></div>
```

上面的语法表示 red 这个 class 存在与否将取决于数据属性 isRed 的真假。

- 数组语法

我们可以把一个数组传给 v-bind:class，以应用一个 class 列表

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

```
const vm = new Vue({
  el: "#app",
  data: {
    classA: 'red',
    classB: 'green'
  }
})
```

```
<!-- 结果 -->
<div id="app" class="red green"></div>
```

- 在数组语法总可以使用三元表达式来切换class

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

```
const vm = new Vue({
  el: "#app",
  data: {
    isActive: true,
    activeClass: 'active',
    errorClass: 'error'
  }
})
```

```
<div id="app" class="active error"></div>
```

- 在数组语法中可以使用对象语法

```
<div id="app" :class="[classA,{green:is}]"></div>
```

```
const vm = new Vue({
  el: "#app",
  data: {
    classA: 'red',
    is: false
  }
})
```

```
<div id="app" class="red"></div>
```

- v-bind:class 可以与普通 class 共存

```
<!-- 这句解析完 red会被直接加在class里 -->
<div v-bind:class="classA" class="red">
<!-- 解析完结果 -->
<div class="...,red">
```

- 绑定style

- 使用对象语法

看着比较像CSS，但其实是一个JavaScript对象

CSS属性名可以用驼峰式(camelCase)或者短横线分隔(kebab-case)来命名

但是使用短横线分隔时，要用引号括起来

```
<div id="app" :style="{background-color:color}">赵拜拜</div>
<!-- 等于 -->
<div id="app" :style="{backgroundColor:color}">赵拜拜</div>
<!-- 结果 -->
<div id="app" style="background-color: red;">赵拜拜</div>
```

```

const vm = new Vue({
  el: "#app",
  data: {
    color: 'red'
  }
})

```

也可以直接绑定一个样式对象，这样模板会更清晰：

```

<div id="app" v-bind:style="objStyle">赵拜拜</div>
<!-- 结果 -->
<div id="app" style="width: 100px; height: 100px; background-color: yellow;">
  赵拜拜
</div>

const vm = new Vue({
  el: "#app",
  data: {
    objStyle: {
      width: '100px',
      height: '100px',
      backgroundColor: 'yellow'
    }
  }
})

```

▪ 使用数组语法

数组语法可以将多个样式对象应用到同一个元素

```

<div id="app" :style="[objStyle1,objStyle2]">赵拜拜</div>
<!-- 结果 -->
<div id="app" style="width: 100px; background-color: yellow; height: 200px;">
  赵拜拜
</div>

const vm = new Vue({
  el: "#app",
  data: {
    objStyle1: {
      width: '100px',
      backgroundColor: 'yellow'
    },
    objStyle2: {
      height: '200px',
    }
  }
})

```

▪ 自动添加前缀

绑定style时，使用需要添加浏览器引擎前缀的CSS属性时，如 transform，Vue.js会自动侦测并添加相应的前缀。

▪ 多重值

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：

```
<div v-bind: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。

• 缩写：：

• 修饰符：

修饰符 (modifier) 是以英文句号 . 指明的特殊后缀，用于指出一个指令应该以特殊方式绑定。

- .camel

由于Vue绑定特性时，会将大写字母转换为小写字母，如：

```

<!-- 最终渲染的结果为： -->
<svg viewBox="0 0 100 100"></svg>

```

所以，Vue提供了v-bind修饰符 camel，该修饰符允许在使用 DOM 模板时将 v-bind 属性名称驼峰化，例如 SVG 的 viewBox 属性(就是把横杠后的单词第一个字符大写)

```
<svg id='app' :viewBox.camel="viewBox"></svg>
<!-- 结果 -->
<svg id='app' viewBox="0 0 100 100"></svg>
```

```
const vm = new Vue({
  el: "#app",
  data: {
    viewBox: '0 0 100 100'
  }
})
```

- `.prop`

被用于绑定 DOM 属性 (property)

```
<div id="app" :text-content.prop="text"></div>
<!-- 结果 -->
<div id="app">hello</div>
```

```
const vm = new Vue({
  el: "#app",
  data: {
    text: 'hello'
  }
})
```

- `.sync`

讲解组件时再说

v-on 指令

- `v-on` 指令可以监听 DOM 事件，并在触发时运行一些 JavaScript 代码
- 事件类型由参数指定

```
<div id="app">
  <button v-on:click="counter += 1">点击加 1</button>
  <p>按钮被点击了 {{ counter }} 次</p>
</div>
```

```
const vm = new Vue({
  el: 'app',
  data: {
    counter: 0
  }
})
```

- 但是很多事件处理逻辑是非常复杂的，所以直接把 JavaScript 代码写在 `v-on` 指令中是不可行的。所以 `v-on` 还可以接收一个需要调用的方法名称。

```
<div id="app">
  <!-- `addCounter` 是在下面定义的方法名 -->
  <button v-on:click="addCounter">点击加 1</button>
  <p>按钮被点击了 {{ counter }} 次</p>
</div>
```

```

const vm = new Vue({
  el: '#app',
  data: {
    counter: 0
  },
  // 在 methods 对象中定义方法
  methods: {
    addCounter: function (e) {
      // this 在方法里指向当前 Vue 实例
      this.counter += 1;
      // e 是原生 DOM 事件
      console.log(e.target);
    }
  }
})

```

- methods中的函数，也会直接代理给Vue实例对象，所以可以直接运行：

```
vm.addCounter();
```

- 除了直接绑定到一个方法，也可以在内联JavaScript语句中调用方法：

```

<div id="app">
  <button v-on:click="addCounter(5)">点击加 5</button>
  <p>按钮被点击了 {{ counter }} 次</p>
</div>

```

```

new Vue({
  el: '#app',
  data: {
    counter: 0
  },
  methods: {
    addCounter: function (num) {
      this.counter += num;
    }
  }
})

```

- 在内联语句中使用事件对象时，可以利用特殊变量 \$event：

```

<div id="app">
  <button v-on:click="addCounter(5, $event)">点击加 5</button>
  <p>按钮被点击了 {{ counter }} 次</p>
</div>

```

```

new Vue({
  el: '#app',
  methods: {
    addCounter: function (num, e) {
      this.counter += 5;
      console.log(e.target);
    }
  }
})

```

- 可以绑定动态事件，Vue版本需要2.6.0+

```
<div v-on:[event]="handleClick">点击，弹出1</div>
```

```

const vm = new Vue({
  el: '#app',
  data: {
    event: 'click'
  },
  methods: {
    handleClick () {
      alert(1);
    }
  }
})

```

- 可以不带参数绑定一个对象，Vue版本需要2.4.0+。

- { 事件名: 事件执行函数 }
- 使用此种方法不支持函数传参&修饰符

```
<div v-on="{ mousedown: doThis, mouseup: doThat }"></div>
```

- v-on指令简写： @

为什么在 HTML 中监听事件？

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

v-on指令的修饰符

事件修饰符

.stop

- 调用 event.stopPropagation，阻止事件冒泡 冒泡：自下向上（自里朝外）

```
<!-- 此时只弹出button -->
<div id="app">
  <div @click="alert('div')">
    <button @click.stop="alert('button')">点击</button>
  </div>
</div>

const vm = new Vue({
  el: '#app',
  methods: {
    alert(str) { alert(str); }
  }
})
```

.prevent

- 调用 event.preventDefault()，阻止默认事件

```
<!-- 点击提交按钮后，页面不会重载 -->
<div id="app">
  <form v-on:submit.prevent="onSubmit">
    <input type="submit">
  </form>
  <!-- 也可以只有修饰符 -->
  <form v-on:submit.prevent>
    <input type="submit">
  </form>
</div>
```

```
const vm = new Vue({
  el: '#app',
  methods: {
    onSubmit() { console.log('submit'); }
  }
})
```

.capture

- 事件捕获模式 捕获：自上向下（自外向里）与冒泡相反

```

<!-- 此时先弹出div再弹出button -->
<div id="app">
  <div @click.capture="alert('div')">
    <button @click="alert('button')">点击</button>
  </div>
</div>

const vm = new Vue({
  el: '#app',
  methods: {
    alert(str) { alert(str) }
  }
})

```

.self

- 只当事件是从侦听器绑定的元素本身触发时才触发回调，只会触发绑定本身的事件

```

<!-- 点击button时，只弹出 button -->
<div id="app">
  <div id="app">
    <div :style="{ backgroundColor: 'red' }"
      @click.self="alert('div')">
      <button @click="alert('button')">点击</button>
    </div>
  </div>
</div>

```

```

const vm = new Vue({
  el: '#app',
  methods: {
    alert(str) { alert(str) }
  }
})

```

.once

- 只触发一次回调
- 2.1.4新增

点击两次button按钮，只弹出一次button

```

<div id="app">
  <button @click.once="alert('button')">点击</button>
</div>

```

```

const vm = new Vue({
  el: '#app',
  methods: {
    alert(str) { alert(str) }
  }
})

```

.passive

- .passive: 告诉浏览器，不会调用"阻止默认事件"
- 设置 addEventListener 中的 passive 选项
- 能够提升移动端的性能
- 2.3.0新增

why passive?

- 即使在触发触摸事件时，执行了一个空的函数，也会让页面卡顿。因为浏览器不知道监听器到底会不会阻止默认事件，所以浏览器要等到执行完整个函数后，才能决定是否要滚动页面。passive事件监听器，允许开发者告诉浏览器，监听器不会阻止默认行为，从而浏览器可以放心大胆的滚动页面，这样可以大幅度提升移动端页面的性能，因为据统计只有20%的触摸事件会阻止默认事件。
- .passive 会告诉浏览器你不想阻止事件的默认行为

注意

1. 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，
`v-on:click.prevent.self` 会阻止所有的点击的默认事件
`v-on:click.self.prevent` 只会阻止对元素自身点击的默认事件
2. 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。

按键修饰符

在监听键盘事件时，我们经常需要检查详细的按键。Vue 允许为 `v-on` 在监听键盘事件时添加按键修饰符

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

你可以直接将 `KeyboardEvent.key` 暴露的任意有效按键名转换为 kebab-case 来作为修饰符。

```
<input v-on:keyup.page-down="onPageDown">
```

在上述示例中，处理函数只会在 `$event.key` 等于 `PageDown` 时被调用。

按键码

使用 `keyCode` 特性也是允许的：

```
<!-- 按回车键会触发执行submit函数 -->
<input v-on:keyup.13="submit">
```

注意：`keyCode` 的事件用法已经被废弃了，并可能不会被最新的浏览器支持。

为了在必要的情况下支持旧浏览器，Vue 提供了绝大多数常用的按键码的别名：

- `.enter` (回车键)
- `.tab`
- `.delete` (捕获“删除”和“退格”键)
- `.esc`
- `.space` (空格键)
- `.up` (箭头上键)
- `.down` (箭头下键)
- `.left` (箭头左键)
- `.right` (箭头右键)

除了使用Vue提供的按键别名之外，还可以自定义按键别名：

```
// 全局配置
// 可以使用 `v-on:keyup.f1`
Vue.config.keyCode.f1 = 112
```

```
Vue.config.keyCode = {
  v: 86,
  f1: 112,
  // 小驼峰 不可用
  mediaPlayPause: 179,
  // 取而代之的是 短横线分隔 且用双引号括起来
  "media-play-pause": 179,
  up: [38, 87]
}
```

```
<input type="text" @keyup.media-play-pause="method">
```

系统修饰键

可以用如下修饰符来实现仅在按下相应按键时才触发鼠标或键盘事件的监听器。

修饰键与常规按键不同，在和 `keyup` 事件一起用时，事件触发时修饰键必须处于按下状态，换句话说，只有在按住 `ctrl` 的情况下释放其它按键，才能触

发 keyup.ctrl。而单单释放 ctrl 也不会触发事件。如果你想要按ctrl键就执行函数，请把 ctrl 换用 keyCode: keyup.17。

- .ctrl
- .alt
- .shift
- .meta

在 Mac 系统键盘上，meta 对应 command 键 (⌘)。

在 Windows 系统键盘 meta 对应 Windows 徽标键 (⊞)。

在 Sun 操作系统键盘上，meta 对应实心宝石键 (◆)。

在其他特定键盘上，尤其在 MIT 和 Lisp 机器的键盘、以及其后继产品，比如 Knight 键盘、space-caDET 键盘，meta 被标记为“META”。

在 Symbolics 键盘上，meta 被标记为“META”或者“Meta”

```
<!-- Alt + C -->
<input @keyup.alt.67="clear">
```

```
<!-- Ctrl + Click -->
<div @click.ctrl="doSomething">Do something</div>
```

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。
- 2.5.0 +

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

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

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

鼠标按钮修饰符

- 仅当点击特定的鼠标按钮时会处理执行函数
- 2.2.0 +
- .left
- .right
- .middle

列表渲染

利用v-for指令，基于数据多次渲染元素。

在v-for中使用数组

用法: (item, index) in items

参数: items: 源数据数组

 item: 数组元素别名

 index: 可选, 索引

可以访问所有父作用域的属性

```
<ul id="app">
  <li v-for="(person, index) in persons">
    {{ index }}--{{ person.name }}--{{ person.age }}
  </li>
</ul>
```

```
const vm = new Vue({
  el: '#app',
  data: {
    persons: [
      { name: '杉杉', age: 18 },
      { name: '思彤哥', age: 20 },
      { name: '成哥', age: 22 },
      { name: '邓哥', age: 88 },
    ]
  }
})
```

可以利用 `of` 替代 `in` 作为分隔符，因为它更接近迭代器的语法：

```
<div v-for="item of items"></div>
```

在v-for中使用对象

用法: `(value, key, index) in Object`

参数: `value`: 对象值

`key`: 可选, 键名

`index`: 可选, 索引

```
<ul id="app">
  <li v-for="(value, key, index) in shan">
    {{ value }}
  </li>
</ul>
```

```
const vm = new Vue({
  el: '#app',
  data: {
    shan: {
      name: '杉',
      age: 18,
      height: '163cm'
    }
  }
})
```

在v-for中使用数字

用法: `n in num`

参数: `n`: 数字, 从1开始

```
<div>
  <span v-for="n in num">{{ n }}</span>
</div>
```

```
const vm = new Vue({
  el: '#app',
  data: {
    num: 10
  }
})
```

在v-for中使用字符串

用法: `str in string`

参数: `str`: 字符串, 源数据字符串中的每一个

```
<div>
  <span v-for="str in string">{{ str }} </span>
</div>
```

```
const vm = new Vue({
  el: '#app',
  data: {
    string: 'shanshan'
  }
})
```

循环一段包含多个元素的内容

可以利用template元素循环渲染一段包含多个元素的内容

```
<ul id="app">
  <template v-for="person in persons">
    <li>{{ person }}</li>
    <li>哈哈</li>
  </template>
</ul>
```

```
const vm = new Vue({
  el: '#app',
  data: {
    persons: ['shan', 'jc', 'cst', 'deng']
  }
})
```

关于key

Vue更新使用v-for渲染的元素列表时，它默认使用“就地更新”的策略。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：

```
<ul id="app">
  <li v-for="(person, index) in persons">
    {{ person }}
    <input type="text" />
    <button @click="handleClick(index)">下移</button>
  </li>
</ul>
```

```
const vm = new Vue({
  el: '#app',
  data: {
    persons: ['shan', 'jc', 'cst', 'deng']
  },
  methods: {
    handleClick (index) {
      const deleteItem = this.persons.splice(index, 1);
      this.persons.splice(index + 1, 0, ...deleteItem);
    }
  }
})
```

在“就地复用”策略中，点击按钮，输入框不随文本一起下移，是因为输入框没有与数据绑定，所以vue.js默认使用已经渲染的dom，然而文本是与数据绑定的，所以文本被重新渲染。这种处理方式在vue中是默认的列表渲染策略，因为高效。

这个默认的模式是高效的，但是在更多的时候，我们并不需要这样去处理，所以，为了给Vue一个提示，以便它能跟踪每个节点的身份，从而重用和重新排序现有元素，我们需要为每项提供一个唯一key特性，Vue会基于 key 的变化重新排列元素顺序，并且会移除 key 不存在的元素。

key的使用方法

预期值: number | string

key不可以再template语句里使用，可以在template子级语句中使用。

有相同父元素的子元素必须有独特的 key，重复的 key 会造成渲染错误，key应唯一。

```
<ul id="app">
  <li v-for="(person, index) in persons" :key="person">
    {{ person }}
  </li>
</ul>
```

```
const vm = new Vue({
  el: '#app',
  data: {
    persons: ['杉杉', '思彤哥', '成哥', '邓哥']
  }
})
```

不建议将数组的索引作为key值，如：

```
<li v-for="(person, index) in persons" :key="index">
  {{ person }}
</li>
```

当改变数组时，页面会重新渲染，Vue会根据key值来判断要不要移动元素。例如当页面重新渲染时，key值为"杉杉"的元素为 `杉杉`，页面重新渲染前，key值为"杉杉"的元素也为 `杉杉`，那么Vue就会移动这个 li 元素，而不是重新生成一个元素。

当使用数组的索引作为key值时，页面重新渲染后，元素的key值会重新被赋值，例如我们将数组进行反转，

反转前：

元素	key值
杉杉	0
思彤哥	1
成哥	2
邓哥	3

反转后：

元素	key值
邓哥	0
成哥	1
思彤哥	2
杉杉	3

Vue会比对渲染前后拥有同样key的元素，发现有变动，就会再生成一个元素，如果用索引作key值得话，那么此时，所有的元素都会被重新生成。

那么key如何唯一的？

跟后台协作时，传回来的每一条数据都有一个id值，这个id就是唯一的，用id做key即可。

key不仅为v-for所有，它可以强制替换元素，而不是重复使用它：

```
<ul id="app">
  <button @click="show = !show">{{ show ? '显示' : '隐藏'}}</button>
  <input type="text" v-if="show" key="a" />
  <input type="text" v-else key="b" />
</ul>
```

```
const vm = new Vue({
  el: '#app',
  data: {
    show: true
  }
})
```

v-for 和 v-if 一同使用

永远不要把 v-if 和 v-for 同时用在同一个元素上。

当 Vue 处理指令时，v-for 比 v-if 具有更高的优先级，所以这个模板：

```
<ul>
  <li
    v-for="user in users"
    v-if="user.isActive"
    :key="user.id"
  >
    {{ user.name }}
  </li>
</ul>
```

将会经过如下运算：

```
this.users.map(function (user) {
  if (user.isActive) {
    return user.name
  }
})
```

因此哪怕我们只渲染出一小部分用户的元素，也得在每次重新渲染的时候遍历整个列表，不论活跃用户是否发生了变化。

所以以下两种场景，我们可以做出如下处理：

1. 为了过滤一个列表中的项目。

```
<ul id="app">
  <li
    v-for="user in users"
    v-if="user.isActive"
    :key="user.id"
  >
    {{ user.name }}
  </li>
</ul>

const vm = new Vue({
  el: '#app',
  data: {
    users: [
      { name: 'shan', isActive: true, id: 1},
      { name: 'jc', isActive: false, id: 2},
      { name: 'cst', isActive: false, id: 3},
      { name: 'deng', isActive: true, id: 4},
    ]
  }
})
```

可以把上面的代码更新为：

```
<!-- 通过原来的数组，得到一个新数组，渲染这个新的数组 -->
<ul>
  <li
    v-for="user in activeUsers"
    :key="user.id"
  >
    {{ user.name }}
  </li>
</ul>
```

```

const vm = new Vue({
  el: '#app',
  data: {
    users: [
      { name: 'shan', isActive: true, id: 1},
      { name: 'jc', isActive: false, id: 2},
      { name: 'cst', isActive: false, id: 3},
      { name: 'deng', isActive: true, id: 4},
    ],
    activeUsers: []
  }
})
vm.activeUsers = vm.users.filter(user => user.isActive);

```

这种方式仅为演示，在日后学习完计算属性后，要利用计算属性来做。

2. 为了避免渲染本应该被隐藏的列表

```

<ul>
  <li
    v-for="user in users"
    v-if="shouldShowUsers"
    :key="user.id"
  >
    {{ user.name }}
  </li>
</ul>

```

```

const vm = new Vue({
  el: '#app',
  data: {
    users: [
      { name: 'shan', isActive: true, id: 1},
      { name: 'jc', isActive: false, id: 2},
      { name: 'cst', isActive: false, id: 3},
      { name: 'deng', isActive: true, id: 4},
    ],
    shouldShowUsers: false
  }
})

```

html部分可替换成为：

```

<ul v-if="shouldShowUsers">
  <li
    v-for="user in users"
    :key="user.id"
  >
    {{ user.name }}
  </li>
</ul>

```

将 `v-if` 置于外层元素上，我们不会再对列表中的每个用户检查 `shouldShowUsers`。取而代之的是，我们只检查它一次，且不会在 `shouldShowUsers` 为否的时候运算 `v-for`。

练习_仿淘宝商品筛选

css文件在文件夹中，自行拷贝

所需数据：

```

goodsList: [
  {
    title: '上装',
    typeList: ['全部', '针织衫', '毛呢外套', 'T恤', '羽绒服', '棉衣', '卫衣', '风衣'],
    id: 1,
  },
  {
    title: '裤装',
    typeList: ['全部', '牛仔裤', '小脚/铅笔裤', '休闲裤', '打底裤', '哈伦裤'],
    id: 2,
  },
  {
    title: '裙装',
    typeList: ['全部', '连衣裙', '半身裙', '长袖连衣裙', '中长款连衣裙'],
    id: 3,
  }
]

```

练习_todoList

css文件在文件夹中，自行拷贝

v-model指令

可以在表单元素上创建双向数据绑定。即数据更新元素更新、元素更新数据也会更新。

本质上v-model为语法糖

元素类型	属性	事件
input[type=text]、textarea	value	input
input[checkbox]、input[radio]	checked	change
select	value	change

input

type=text 文本框

```

<div id="app">
  <input v-model="message">
  <p>Message 为: {{ message }}</p>
</div>

<input id='app' type="text" @input="$event.target.value" :value="msg">

const vm = new Vue({
  el: '#app',
  data: {
    message: ''
  }
})

```

type=checkbox 复选框

单个复选框

绑定到布尔值，v-model="Boolean"

```
<div id="app">
<input
  type="checkbox"
  id="checkbox"
  v-model="checked"
/>
<label for="checkbox">{{ checked }}
```

```
const vm = new Vue({
  el: '#app',
  data: {
    checked: true
  }
})
```

多个复选框

绑定到同一个数组，v-model="Array"
数组中的值为被选中的input框value值

```
<div id="app">
<input type="checkbox" id="cheng" value="成哥" v-model="checkedNames">
<label for="cheng">成哥</label>

<input type="checkbox" id="deng" value="邓哥" v-model="checkedNames">
<label for="deng">邓哥</label>

<input type="checkbox" id="tong" value="思彤哥" v-model="checkedNames">
<label for="tong">思彤哥</label>
<br>
<span>被选中的人有: {{ checkedNames }}</span>
</div>

const vm = new Vue({
  el: '#app',
  data: {
    checkedNames: []
  }
})
```

type=radio 单选框

被绑定的数据和value同步

```
<div id="app">
<input type="radio" id="cheng" value="成哥" v-model="picked">
<label for="cheng">成哥</label>
<input type="radio" id="deng" value="邓哥" v-model="picked">
<label for="deng">邓哥</label>
<input type="radio" id="tong" value="思彤哥" v-model="picked">
<label for="deng">思彤哥</label>
<br>
<span>被选中的人: {{ picked }}</span>
</div>

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

textarea

```
<div id="app">
  <p>多行文本为: {{ message }}</p>
  <textarea v-model="message" placeholder="添加文本"></textarea>
</div>
```

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

select

匹配的值为option中的汉字，如果想匹配value中的值，就在option里加入value

单选

```
<div id="app">
  <select v-model="selected">
    <option>A</option>
    <option>B</option>
    <option>C</option>
  </select>
  <span>选择: {{ selected === '请选择' ? '' : selected }}</span>
</div>
<!-- 匹配value --&gt;
&lt;select v-model="selected"&gt;
  &lt;option value='1'&gt;A&lt;/option&gt;
  &lt;option value='2'&gt;B&lt;/option&gt;
  &lt;option value='3'&gt;C&lt;/option&gt;
&lt;/select&gt;

const vm = new Vue({
  el: '#app',
  data: {
    selected: '请选择'
  }
})</pre>
```

注意：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，可以提供一个值为空的禁用选项：

```
<div id="app">
  <select v-model="selected">
    <option :disabled="selected">请选择</option>
    <option>A</option>
    <option>B</option>
    <option>C</option>
  </select>
  <span>选择: {{ selected === '请选择' ? '' : selected }}</span>
</div>
```

多选

绑定到一个数组

```
<div id="app">
  <select v-model="selected" multiple>
    <option>A</option>
    <option>B</option>
    <option>C</option>
  </select>
  <span>选择: {{ selected }}</span>
</div>
```

```
const vm = new Vue({
  el: '#app',
  data: {
    selected: []
  }
})
```

修饰符

.lazy

在默认情况下，v-model在每次input事件触发后将输入框的值与数据进行同步。如果要变为使用change事件同步可以添加lazy修饰符：

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

.number

自动将用户的输入值转为数值类型：

```
<input v-model.number="age" type="number">
```

.trim

自动过滤用户输入的首尾空白字符：

```
<input v-model.trim="msg">
```

练习_简易计算器

练习_调查问卷

```

questionList: [
  {
    type: 'short',
    title: '1.请问你的姓名是?',
    chooseList: null,
    answer: '',
    id: 0
  },
  {
    type: 'single',
    title: '2.请问您的性别是?',
    chooseList: [
      '男',
      '女',
      '保密'
    ],
    answer: '',
    id: 1,
  },
  {
    type: 'multiple',
    title: '3. 请选择您的兴趣爱好:',
    chooseList: [
      '看书',
      '游泳',
      '跑步',
      '看电影',
      '听音乐'
    ],
    answer: [],
    id: 2,
  },
  {
    type: 'long',
    title: '4. 请介绍一下自己:',
    chooseList: null,
    answer: '',
    id: 3,
  }
]

```

计算属性

有些时候，我们在模板中放入了过多的逻辑，从而导致模板过重，且难以维护。例如：

```

<div id="app">
  {{ message.split('').reverse().join('') }}
</div>

```

碰到这样的情况，我们必须看一段时间才能意识到，这里是想要显示变量`message`的翻转字符串，而且，一旦我们想要在模板中多次使用翻转字符串时，会更加麻烦。

所以，当我们处理复杂逻辑时，都应该使用计算属性。

基础用法

计算属性是Vue配置对象中的属性，使用方式如下：

```

<div id="app">
  <!-- 计算属性的值可以像data数据一样，直接被使用 -->
  {{ someComputed }}
</div>

```

```
const vm = new Vue({
  el: '#app',
  computed: {
    // 返回的值，就是计算属性的值
    someComputed () {
      return 'some values'
    }
  }
})
```

例如，我们想要获取到一串字符串的翻转字符串，我们可以利用计算属性来做：

```
<div id="app">
  <p>原始字符串: "{{ msg }}"</p>
  <p>翻转字符串: "{{ reversedMsg }}"</p>
</div>
```

```
const vm = new Vue({
  el: '#app',
  data: {
    msg: 'Hello'
  },
  computed: {
    reversedMsg: function () {
      return this.msg.split('').reverse().join('');
    }
  }
})
```

我们可以看到，reversedMsg的值取决于msg的值，所以，当我们更改msg的值时，reversedMsg的值也会随之更改。

计算属性 vs 方法

其实，我们上述的功能，利用方法也可以实现，如：

```
<div id="app">
  <p>原始字符串: "{{ msg }}"</p>
  <p>翻转字符串: "{{ reversedMsg() }}"</p>
</div>
```

```
const vm = new Vue({
  el: '#app',
  data: {
    msg: 'Hello'
  },
  methods: {
    reversedMsg: function () {
      return this.msg.split('').reverse().join('');
    }
  }
})
```

虽然在表达式中调用方法也可以实现同样的效果，但是使用 计算属性 和 使用 方法 有着本质的区别。

当使用方法时，每一次页面重新渲染，对应的方法都会重新执行一次，如：

```
<div id="app">
  <p>{{ name }}</p>
  <p>{{ reversedMsg() }}</p>
</div>
```

```

const vm = new Vue({
  el: '#app',
  data: {
    msg: 'Hello',
    name: 'shanshan'
  },
  methods: {
    reversedMsg: function () {
      console.log('方法执行啦');
      return this.msg.split('').reverse().join('');
    }
  }
})
vm.name = 'duyiyi';

```

在上面的例子中我们可以看到，一旦更改name的值，页面会重新渲染，此刻控制台中打印出 方法执行啦 这串字符串，代表着reversedMsg这个函数执行了，但是我们并不需要该方法执行，因为改动的数据和这个函数没有任何关系，如果这个函数内的逻辑很复杂，那么对于性能来讲，也是一种消耗。

但是利用计算属性做，就不会有这样的现象出现，如：

```

const vm = new Vue({
  el: '#app',
  data: {
    msg: 'Hello',
    name: 'shanshan'
  },
  computed: {
    reversedMsg: function () {
      console.log('计算执行啦');
      return this.msg.split('').reverse().join('');
    }
  }
})
vm.name = 'duyiyi';

```

此时可以看到，当给数据name重新赋值时，计算属性并没有执行。

所以，计算属性和方法的本质的区别，是：计算属性是基于响应式依赖进行缓存的，计算属性的值一直存于缓存中，只要它依赖的data数据不改变，每次访问计算属性，都会立刻返回缓存的结果，而不是再次执行函数。而方法则是每次触发重新渲染，调用方法将总会再次执行函数。

那么，为什么需要缓存呢？

假如说，我们有一个计算属性A，它需要遍历一个巨大的数组并且做巨大的计算。然后我们需要使用到这个计算属性A，如果没有缓存，我们就会再次执行A的函数，这样性能开销就变得很大了。

深入计算属性

计算属性除了写成一个函数之外，还可以写成一个对象，对象内有两个属性，`get`&`set`，这两个属性皆为函数，写法如下：

```

const vm = new Vue({
  el: '#app',
  computed: {
    fullName: {
      get () {
        // 一些代码
      },
      set () {
        // 一些代码
      }
    }
  }
})

```

get 读取

在前面，我们直接将计算属性写成了一个函数，这个函数即为`get`函数。也就是说，计算属性默认只有`get`。`get`的`this`，被自动绑定为Vue实例。

使用`{}computed{}`会自动调用`get`方法

何时执行?

当我们去获取某一个计算属性时，就会执行get函数。

```
const vm = new Vue({
  el: '#app',
  data: {
    msg: 'Hello'
  },
  computed: {
    reversedMsg: {
      get () {
        return this.msg.split('').reverse().join('');
      }
    }
  }
})
```

set 设置

可选，set函数在给计算属性重新赋值时会执行。

参数：为被重新设置的值。

set的this，被自动绑定为Vue实例。

当 reverseMsg = 'xxxx'时 set方法会被调用，参数为 'xxxx'

```
const vm = new Vue({
  el: '#app',
  data: {
    msg: 'Hello',
    firstStr: ''
  },
  computed: {
    reversedMsg: {
      get () {
        return this.msg.split('').reverse().join('');
      },
      set (newVal) {
        this.firstStr = newVal[0];
      }
    }
})
```

要注意，即使给计算属性赋了值，计算属性也不会改变，在重复一遍，只有当依赖的响应式属性变化了，计算属性才会重新计算。

练习_姓名筛选

```
personArr: [
  {
    name: '王港',
    src: 'https://ss0.bdstatic.com/70cFvHSh_Q1YnxGkpoWK1HF6hy/it/u=4005587090,2408158268&fm=26&gp=0.jpg',
    des: '颈椎不好',
    sex: 'm',
    id: '056482'
  },
  {
    name: '刘莹',
    src: 'https://timgsa.baidu.com/timg?image&quality=80&size=b9999_10000&sec=1571915784984&di=a0056fd06188e87b922c60878e5ce6e2&imgtype=0&src=htt|',
    des: '我是谁',
    sex: 'f',
    id: '157894'
  },
  {
    name: '刘秀莹',
    src: 'https://timgsa.baidu.com/timg?image&quality=80&size=b9999_10000&sec=1571915803971&di=47dc968f55b16a461de3e8f25bdf8600&imgtype=0&src=htt|',
    des: '你没有见过陌生的脸',
    sex: 'f',
    id: '2849245'
  },
  {
    name: '刘金雷',
    src: 'https://timgsa.baidu.com/timg?image&quality=80&size=b9999_10000&sec=1571915748758&di=5be825da4d37bcc21959946c101d5609&imgtype=0&src=htt|',
    des: '瓜皮刘',
    sex: 'm',
    id: '348515'
  },
  {
    name: '刘飞翔',
    src: 'https://timgsa.baidu.com/timg?image&quality=80&size=b9999_10000&sec=1571915762633&di=49517ca62ecddb638cdfb2158a64e39a&imgtype=0&src=htt|',
    des: ' ',
    sex: 'm',
    id: '478454'
  }
],
```

练习_全选商品

```

courseList: [
  {
    poster: 'https://img.alicdn.com/bao/uploaded/i1/TB1VtAgdlWD3KVjSZFs3KIqkpXa_040950.jpg_80x80.jpg',
    title: '渡一教育 CSS3 深度剖析',
    price: 1299,
    cart: 1,
    id: 0
  },
  {
    poster: 'https://img.alicdn.com/bao/uploaded/i7/TB1_VJecBWd3KVjSZKPagip7FXa_045814.jpg_80x80.jpg',
    title: '渡一教育 移动端开发课程',
    price: 1148,
    cart: 1,
    id: 1595402664708
  },
  {
    poster: 'https://img.alicdn.com/bao/uploaded/i2/TB1J.Q4cQxz61VjSZFto7uDSVXa_010347.jpg_80x80.jpg',
    title: '渡一教育 2019年 HTMLCSS零基础入学宝典',
    price: 1,
    cart: 1,
    id: 1596305473062
  },
  {
    poster: 'https://img.alicdn.com/bao/uploaded/i2/TB1bHwlaCWD3KVjSZSgVbgCxVXa_032434.jpg_80x80.jpg',
    title: '渡一教育 Web前端开发JavaScriptJs课',
    price: 1,
    cart: 1,
    id: 1595413512182
  },
  {
    poster: 'https://img.alicdn.com/bao/uploaded/i2/TB1MJd3g4z1gK0jSZSgnHevwpXa_014447.jpg_80x80.jpg',
    title: 'Web前端开发高级工程师全阶班【渡一教育】',
    price: 12798,
    cart: 1,
    id: 1596302161181
  },
  {
    poster: 'https://img.alicdn.com/bao/uploaded/i6/TB1xPeAbwaH3KVjSZFpjLhKpXa_105848.jpg_80x80.jpg',
    title: '渡一教育 Java零基础入门到精通（集合，泛型等）',
    price: 1,
    cart: 1,
    id: 1596300025301
  },
]

```

侦听器

侦听属性，响应数据（data&computed）的变化，当数据变化时，会立刻执行对应函数，

值类型

函数类型

例：

```

const vm = new Vue({
  el: '#app',
  data: {
    msg: 'hello, 你好呀，我是杉杉',
  },
  watch: {
    msg () {
      console.log('msg的值改变啦~');
    }
  }
})
// 更改msg的值
vm.msg = 'hello~~~'; // 此时会在控制台中打印出` msg的值改变啦 `
```

侦听器函数，会接收两个参数，第一个参数为newVal(被改变的数据)，第二个参数为oldVal(赋值新值之前的值)。如在上述代码中，将侦听器watch更改一下，如：

```
watch: {
  msg (newVal,oldVal) {
    conosle.log(newVal, oldVal);
  }
}

// 更改msg的值
vm.msg = 'hello~~~~'; // 此时会在控制台中打印出`hello, 你好呀，我是杉杉 hello~~~~`
```

字符串类型

值为方法名字，被侦听的数据改变时，会执行该方法。

```
const vm = new Vue({
  el: '#app'
  data: {
    msg: '杉杉'
  },
  watch: {
    msg: 'msgChange'
  },
  methods: {
    msgChange () {
      console.log('msg的值改变啦');
    }
  }
})
vm.msg = 'hello'; // 此时msgChange函数会执行，控制台中打印出 ` msg的值改变啦 `
```

对象类型

写成对象类型时，可以提供选项。

handler

必需。handler时被侦听的数据改变时执行的回调函数。

handler的值类型为函数/字符串，写成字符串时为一个方法的名字。

```
const vm = new Vue({
  el: '#app'
  data: {
    msg: '杉杉'
  },
  watch: {
    msg: {
      handler () {
        console.log('msg的值改变啦');
      }
    }
  }
})
vm.msg = 'hello'; // 此时回调函数会执行，控制台中打印出 ` msg的值改变啦 `
```

deep

在默认情况下，侦听器侦听对象只侦听引用的变化，只有在给对象赋值时它才能被监听到。所以需要使用deep选项，让其可以发现对象内部值的变化，将deep的值设置为true，那么无论该对象被嵌套的有多深，都会被侦听到。

```

const vm = new Vue({
  el: '#app'
  data: {
    personObj: {
      name: '邓旭明',
      age: 88
    }
  },
  watch: {
    personObj: {
      handler () {
        console.log('对象的值改变啦');
      },
      deep: true // 开启深度侦听
    }
  }
})
vm.obj.name = '老邓头'; // 此时回调函数会执行，控制台中打印出`对象的值改变啦`
```

注意，当对象的属性较多的时候，性能开销会比较大，此时可以监听对象的某个属性，这个后面再说。

immediate

加上immediate选项后，回调将会在侦听开始之后立刻被调用。而不是等待侦听的数据更改后才会调用。

```

const vm = new Vue({
  el: '#app'
  data: {
    msg: '杉杉'
  },
  watch: {
    msg: {
      handler () {
        console.log('回调函数执行啦');
      },
      immediate: true
    }
  }
})
// 此时未更改msg的值，就会在控制台打印出来`回调函数执行啦`
```

数组类型

可以将多种不同值类型写在一个数组中。如：

```

const vm = new Vue({
  el: '#app'
  data: {
    msg: '杉杉'
  },
  watch: {
    msg: [
      'msgChange',
      function () {},
      {
        handler () {},
        deep: true,
        immediate: true
      }
    ]
  }
})
```

键类型

正常对象key值

以上演示的都是正常的对象key值，这里不再赘述。

字符串类型key值

当key值类型为字符串时，可以实现监听对象当中的某一个属性，如：

```
const vm = new Vue({
  el: '#app'
  data: {
    personObj: {
      name: '邓旭明',
      age: 88
    }
  },
  watch: {
    'personObj.name' () {
      console.log('对象的值改变啦');
    }
  }
})
vm.obj.name = '老邓头'; // 此时回调函数会执行，控制台中打印出 ' 对象的值改变啦 '
```

vm.\$watch

Vue实例将会在实例化时调用\$watch，遍历watch对象的每一个属性。

我们也可以利用vm.\$watch来实现侦听，用法与watch选项部分一致，略有不同。以下为使用方法。

1. 侦听某个数据的变化

```
// 1. 三个参数，一参为被侦听的数据；二参为数据改变时执行的回调函数；三参可选，为设置的选项对象
vm.$watch(
  'msg',
  function () {
    // 干了点事儿
  },
  {
    deep: Boolean,
    immediate: Boolean
  }
)

// 2. 二个参数，一参为被侦听的数据；二参为选项对象，其中handler属性为必需，是数据改变时执行的回调函数，其他属性可选。
vm.$watch(
  'msg',
  {
    handler () {
      // 干了点事儿
    },
    deep: Boolean,
    immediate: Boolean
  }
)
```

2. 侦听某个对象属性的变化

```
vm.$watch('obj.name', /*参数和上面一之*/)
```

3. 当监听的数据的在初始不确定，由多个数据得到时，此时可以将第一个参数写成函数类型

```
vm.$watch(function () {
  // 表达式`this.a + this.b`每次得出一个不同的结果时该函数都会被调用
  // 这就像监听一个未被定义的计算属性
  return this.a + this.b;
}, /*参数和上面一致*/)
```

侦听器函数执行后，会返回一个取消侦听函数，用来停止触发回调：

```
const unwatch = vm.$watch('msg', function () {});
unwatch(); // 执行后会取消侦听msg数据
```

使用unwatch时，需要注意的是，在带有immediate选项时，不能在第一次回调时取消侦听数据。

```
const unwatch = vm.$watch('msg', function () {
  // 干了点儿事
  unwatch(); // 此时会报错
}, {
  immediate: true
})
```

如果仍然希望在回调内部用一个取消侦听的函数，那么可以先检查该函数的可用性：

```
var unwatch = vm.$watch('msg', function () {
  // 干了点儿事
  if(unwatch) {
    unwatch();
  }
}, {
  immediate: true
})
```

侦听器 vs 计算属性

0. 侦听器不写immediate时在属性改变时才会启动，只要侦听器启动，它会在计算属性之前执行
1. 两者都可以观察和响应Vue实例上的数据的变动。
2. watch擅长处理的场景是：一个数据影响多个数据。计算属性擅长处理的场景是：多个数据影响一个数据。
3. 在侦听器中可以执行异步，但是在计算属性中不可以，例：

使用侦听器：

```
var vm = new Vue({
  el: '#app',
  data: {
    question: '',
  },
  watch: {
    question () {
      setTimeout(() => {
        alert(this.question);
      }, 1000)
    }
  }
})
```

练习_仿百度搜索联想

url: <https://sp0.baidu.com/5a1Fazu8AA54nxGko9WTAnF6hh/su>

请求方式(服务器返回的数据类型): jsonp

发送参数:

1. wd: 字符串，搜索的文字
2. cb: 字符串，callback函数的名字

返回结果: (JSON格式)

```
{
  q: String,
  p: Boolean,
  s: Array // 搜索联想列表
}
```

vue-resource

在Vue中实现异步加载需要使用到vue-resource库，利用该库发送ajax。

引入vue-resource

```
<script src="https://cdn.jsdelivr.net/npm/vue-resource@1.5.1"></script>
```

要注意的是，vue-resource依赖于Vue，所以要先引入Vue，再引入vue-resource。

引入vue-resource之后，在Vue的全局上会挂载一个\$http方法，在vm.\$http方法上有一系列方法，每个HTTP请求类型都有一个对应的方法。

vue-resource使用了promise，所以\$http中的方法的返回值是一个promise。

请求方法

POST请求

用于提交数据

常用data格式：

- 表单提交：multipart/form-data，比较老的网站会使用表单提交去获取数据，现在基本都不用表单提交，而是使用ajax，但是现在表单提交仍然存在，有时候需要做图片上传、文件上传。
- 文件上传：application/json，现在大多数情况下都是用这个格式

使用方法：vm.\$http.post(url, [body], [options])

- url：必需，请求目标url
- body：非必需，作为请求体发送的数据
- options：非必需，作为请求体发送的数据

```
this.$http.post('https://developer.duyiedu.com/vue/setUserInfo', {
  name: this.name,
  mail: this.mail
})
.then(res => {
  console.log(res);
})
.catch(error => {
  console.log(error);
})
```

GET请求

获取数据

使用方法：vm.\$http.get(url, [options])

```
this.$http.get('https://developer.duyiedu.com/vue/getUserInfo')
.then(res => {
  console.log(res);
})
.catch(error => {
  console.log(error);
})
```

在get请求时传参：

```

this.$http.get('https://developer.duyiedu.com/vue/getUserInfo', {
  params: {
    id: 'xxx'
  }
})
.then(res => {
  console.log(res);
})
.catch(error => {
  console.log(error);
})

```

PUT请求

更新数据，将所有的数据全都推送到后端

使用方法: `vm.$http.put(url, [body], [config])`

PATCH请求

更新数据，只将修改的数据全都推送到后端

使用方法: `vm.$http.patch(url, [body], [config])`

DELETE请求

删除数据

使用方法: `vm.$http.delete(url, [config])`

HEAD请求

请求头部信息

使用方法: `vm.$http.head(url, [config])`

JSONP请求

除了jsonp以外，以上6种的API名称是标准的HTTP方法。

使用方法: `vm.$http.jsonp(url, [options]);`

```

this.$http.jsonp('https://developer.duyiedu.com/vue/jsonp').then(res => {
  this.msg = res.bodyText;
});

this.$http.jsonp('https://sp0.baidu.com/5a1Fazu8AA54nxGko9WTAnF6hh/su', {
  params: {
    wd: 'nn',
  },
  jsonp: 'cd', //jsonp默认是callback，百度缩写成了cb，所以需要指定下
})
.then(res => {
  //这相当于正确信息的回调函数
  console.log(res);
})

```

options 参数说明

参数	类型	描述
<code>url</code>	<code>String</code>	请求目标url
<code>body</code>	<code>Object, FormData, string</code>	作为请求体发送的数据
<code>headers</code>	<code>Object</code>	作为请求头部发送的头部对象

参数	类型	描述
params	Object	作为URL参数的参数对象
method	String	HTTP方法 (例如GET, POST, ...)
responseType	String	设置返回数据的类型
timeout	Number	在请求发送之前修改请求的回调函数
credentials	Boolean	是否需要出示用于跨站点请求的凭据
emulateHTTP	Boolean	是否需要通过设置X-HTTP-Method-Override头部并且以传统POST方式发送PUT, PATCH和DELETE请求。
emulateJSON	Boolean	设置请求体的类型为application/x-www-form-urlencoded
before	function(request)	在请求发送之前修改请求的回调函数
uploadProgress	function(event)	用于处理上传进度的回调函数
downloadProgress	function(event)	用于处理下载进度的回调函数

响应对象

通过如下属性和方法处理一个请求获取到的响应对象：

属性

属性	类型	描述
url	String	响应的 URL 源
body	Object, Blob, string	响应体数据
headers	Header	请求头部对象
ok	Boolean	当 HTTP 响应码为 200 到 299 之间的数值时该值为 true
status	Number	HTTP 响应码
statusText	String	HTTP 响应状态

方法

方法	描述
text()	以字符串方式返回响应体
json()	以格式化后的 json 对象方式返回响应体
blob()	以二进制 Blob 对象方式返回响应体

以json()为例：

```
this.$http.get('https://developer.duyiedu.com/vue/getUserInfo')
  .then(res => {
    return res.json();
  })
  .then(res => {
    console.log(res);
  })
```

最后的话

很不幸，Vue官方已不再维护这个库了，so...哈哈哈，我们再学点其他的[* ↴ ↵ *]9

Axios

Axios是一个基于promise的HTTP库

浏览器支持情况: Chrome、Firefox、Safari、Opera、Edge、IE8+

引入

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

API

- `axios(config)`
- `axios(url, [config])`

config 配置对象

最常用的配置:

```
axios({
  method: 'get', // post、get、put....
  baseURL: '', // 请求的域名，基本地址
  url: '', // 请求的路径
  params: {}, // 会将请求参数拼接在url上
  data: {}, // 会将请求参数放在请求体中
  headers: {}, // 设置请求头，例如设置token等
  timeout: 1000, // 设置请求超时时长，单位: ms
})
```

方法别名

为方便起见，为所有支持的请求方法提供了别名。

- `axios.request(config)`
- `axios.get(url, [config])`
- `axios.post(url, [data], [config])`
- `axios.delete(url, [config])`
- `axios.head(url, [config])`
- `axios.put(url, [data], [config])`
- `axios.patch(url, [data], [config])`
- `axios.options(url, [config])`

配置默认值

可以指定将被用在各个请求的配置默认值

全局配置

```
axios.defaults.baseURL = 'https://developer.duyiedu.com/vue';
axios.defaults.timeout = 1000;
```

在实际项目中，很少用全局配置。

实例配置

可以使用自定义配置新建一个axios实例

```
const instance = axios.create({
  baseURL: 'https://developer.duyiedu.com/vue',
  timeout: 1000,
})

instance.get('/getUserInfo').then(res => {
  // ...
})
```

请求配置

```
const instance = axios.create();
instance.get('/getUserInfo', {
  timeout: 5000
})
```

配置的优先顺序

全局 < 实例 < 在axios请求本身上

并发

同时进行多个请求，并统一处理返回值

- axios.all(iterable)
- axios.spread(callback)

```
axios.all([
  axios.get('/a'),
  axios.get('/b')
]).then(axios.spread((aRes, bRes) => {
  console.log(aRes, bRes);
}))
```

拦截器

interceptors，在发起请求之前做一些处理，或者在响应回来之后做一些处理。

请求拦截器

```
axios.interceptors.request.use(config => {
  // 在发送请求之前做些什么
  return config;
})
```

响应拦截器

```
axios.interceptors.response.use(response => {
  // 对响应数据做点什么
  return response;
})
```

移除拦截器

```
const myInterceptor = axios.interceptors.request.use(config => {});
axios.interceptors.request.eject(myInterceptor);
```

为axios实例添加拦截器

```
const instance = axios.create();
instance.interceptors.request.use(config => {});
```

取消请求

用于取消正在进行的http请求

```
const source = axios.CancelToken;
const source = CancelToken.source();

axios.get('/getUserInfo', {
  cancelToken: source.token
}).then(res => {
  console.log(res);
}).catch(error => {
  if(axios.isCancel(error)) {
    // 取消请求
    console.log(error.message);
  } else {
    // 处理错误
  }
})

// 取消请求 参数 可选
source.cancel('取消请求');
```

错误处理

在请求错误时进行的处理

request / response 是error的上下文，标志着请求发送 / 得到响应
在错误中，如果响应有值，则说明是响应时出现了错误。
如果响应没值，则说明是请求时出现了错误。
在错误中，如果请求无值，则说明是请求未发送出去，如取消请求。

```
axios.get('/user/12345')
  .catch(function (error) {
    // 错误可能是请求错误，也可能是响应错误
    if (error.response) {
      // 响应错误
    } else if (error.request) {
      // 请求错误
    } else {
      console.log('Error', error.message);
    }
    console.log(error.config);
 });
```

在实际开发过程中，一般在拦截器中统一添加错误处理

请求拦截器中的错误，会当请求未成功发出时执行，但是要注意的是：取消请求后，请求拦截器的错误函数也不会执行，因为取消请求不会抛出异常，`axios`对其进行了单独的处理。

在更多的情况下，我们会在响应拦截器中处理错误。

```
const instance = axios.create({});

instance.interceptors.request(config => {

}, error => {
  return Promise.reject(error);
})

instance.interceptors.response(response => {

}, error => {
  return Promise.reject(error);
})
```

axios 预检

当`axios`的请求为非简单请求时，浏览器会进行预检，及发送`OPTIONS`请求。请求到服务器，询问是否允许跨域。如果响应中允许预检中请求的跨域行为，则浏览器会进行真正的请求。否则会报`405`错误。

template 选项

关于el

提供一个在页面上已存在的 DOM 元素作为 Vue 实例的挂载目标。可以是 CSS 选择器，也可以是一个 HTML 元素 实例。

如果在实例化时存在这个选项，实例将立即进入编译过程，否则，需要显式调用 `vm.$mount()` 手动开启编译。

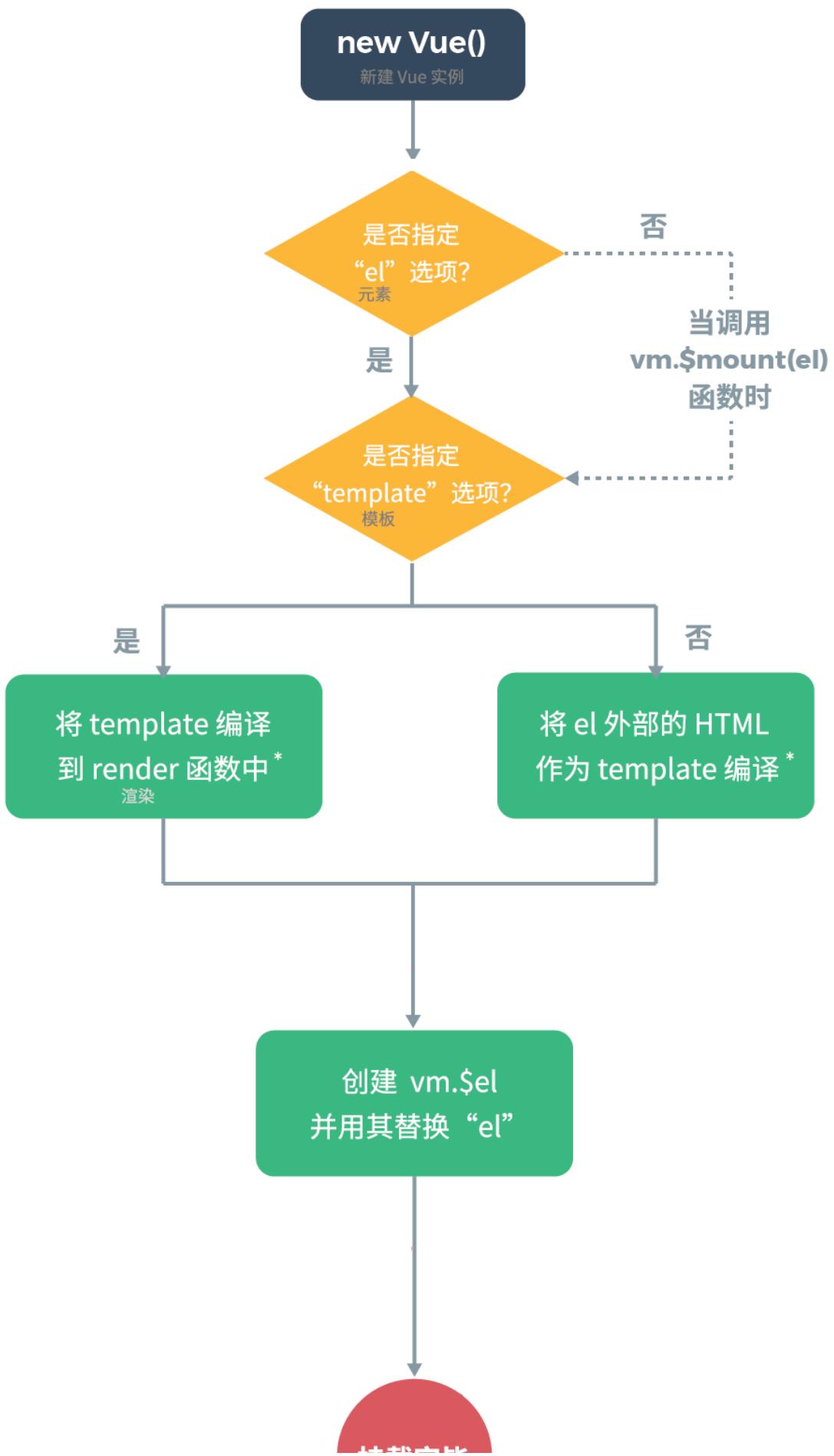
template

一个字符串模板作为 Vue 实例的标识使用。模板将会 替换 挂载的元素，挂载元素的内容都将被忽略。

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

const vm = new Vue({
  el: '#app',
  template:
    `<div id="ceshi">xxx</div>
  `,
})
```

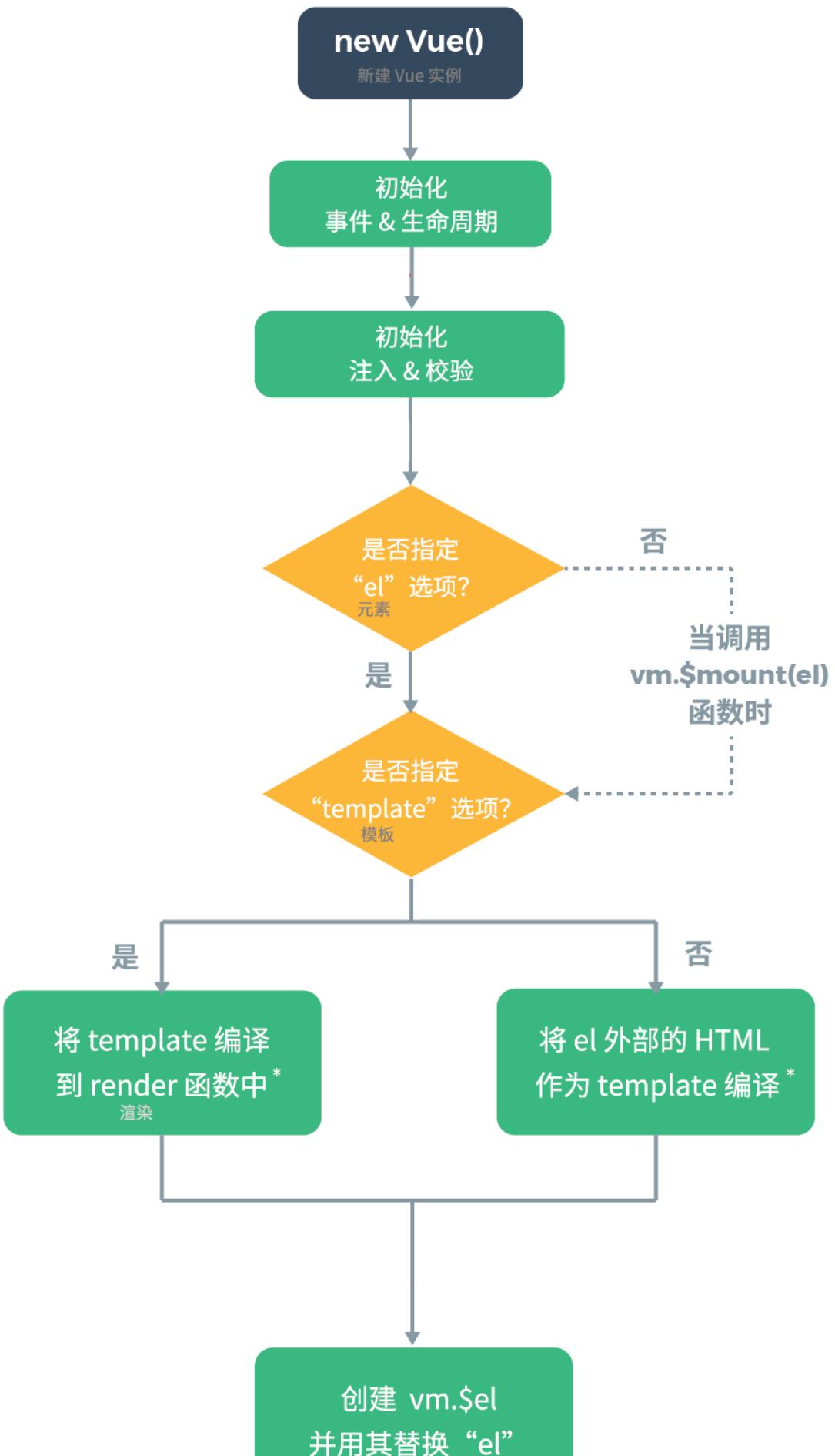
Vue初始化到挂载的流程

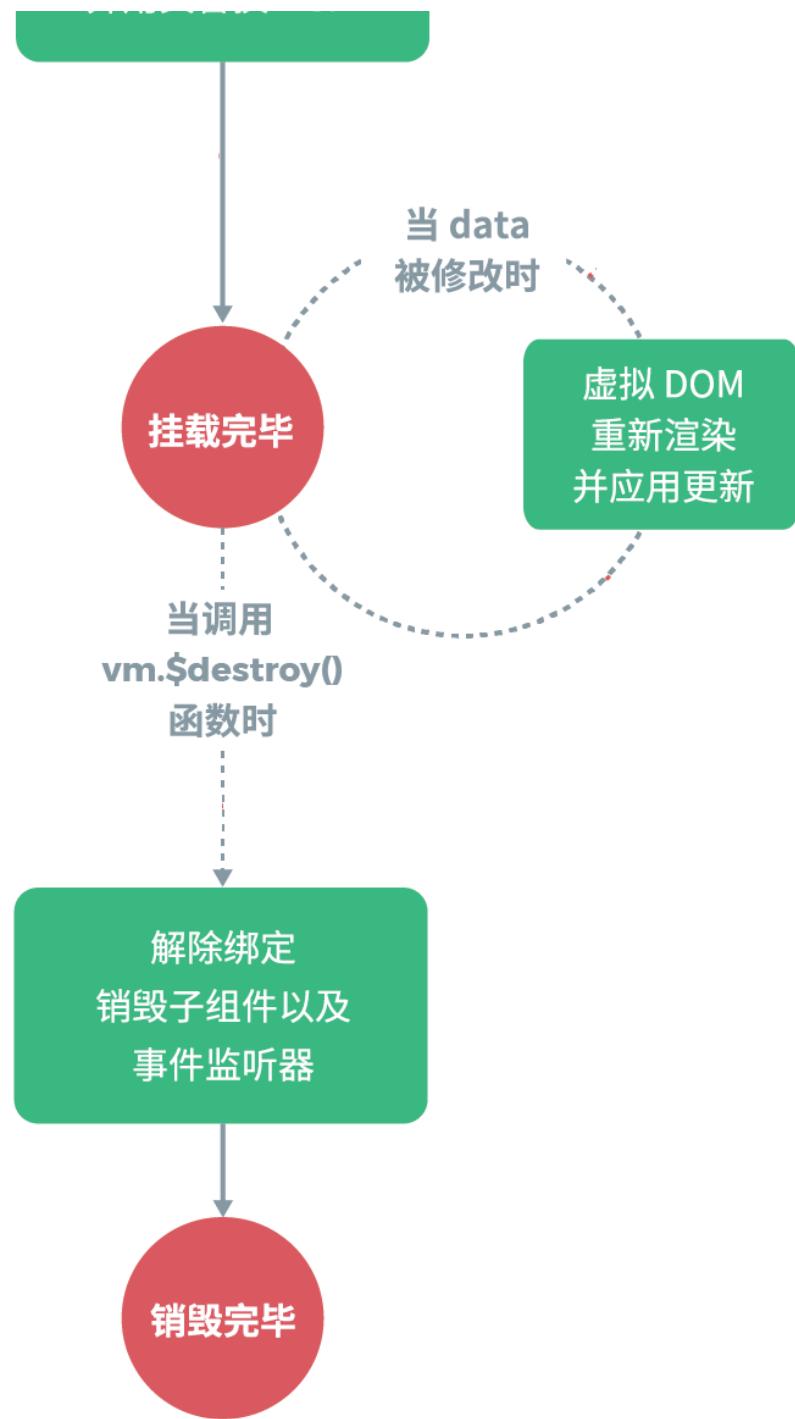


Vue生命周期

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。

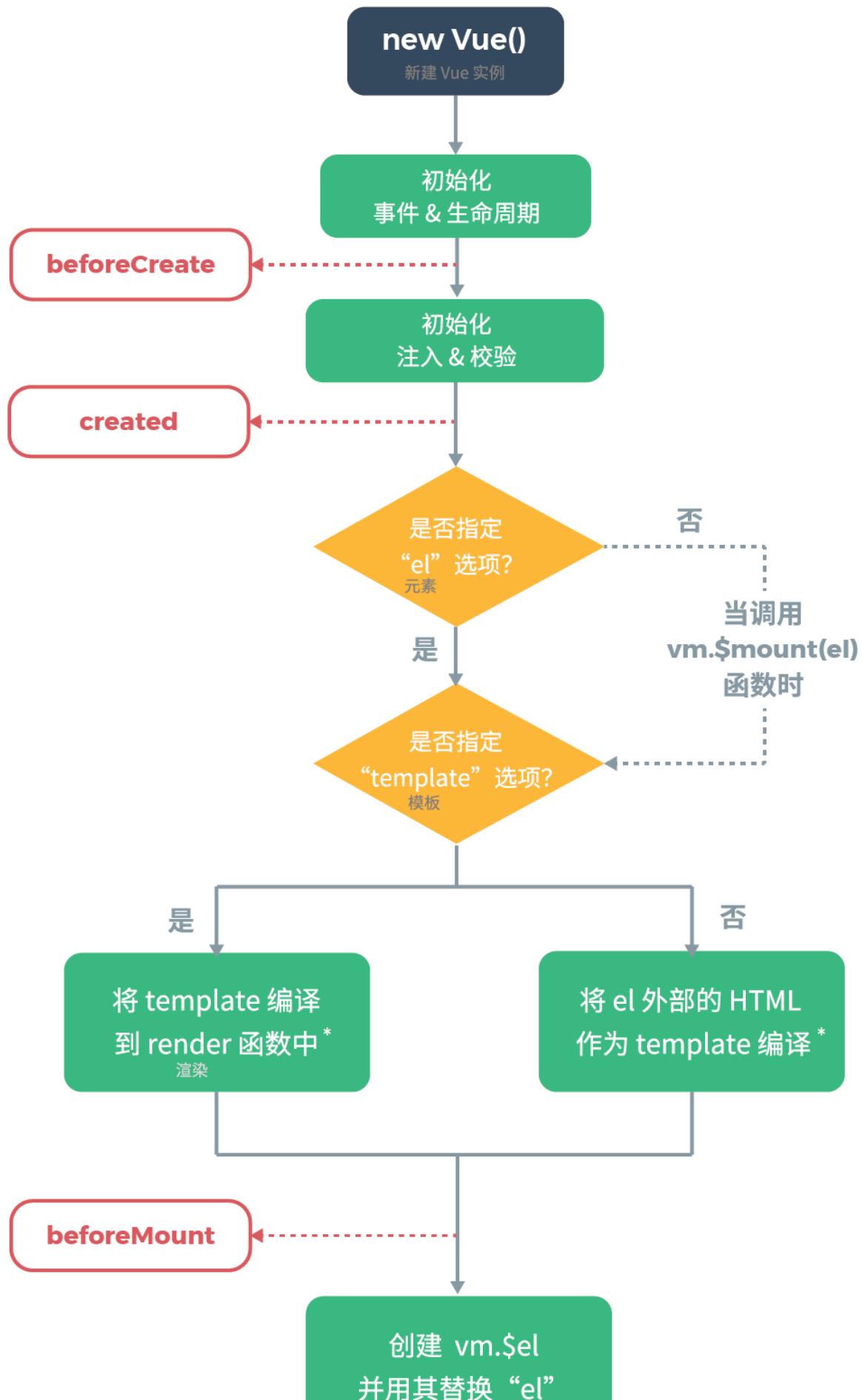
生命周期图示

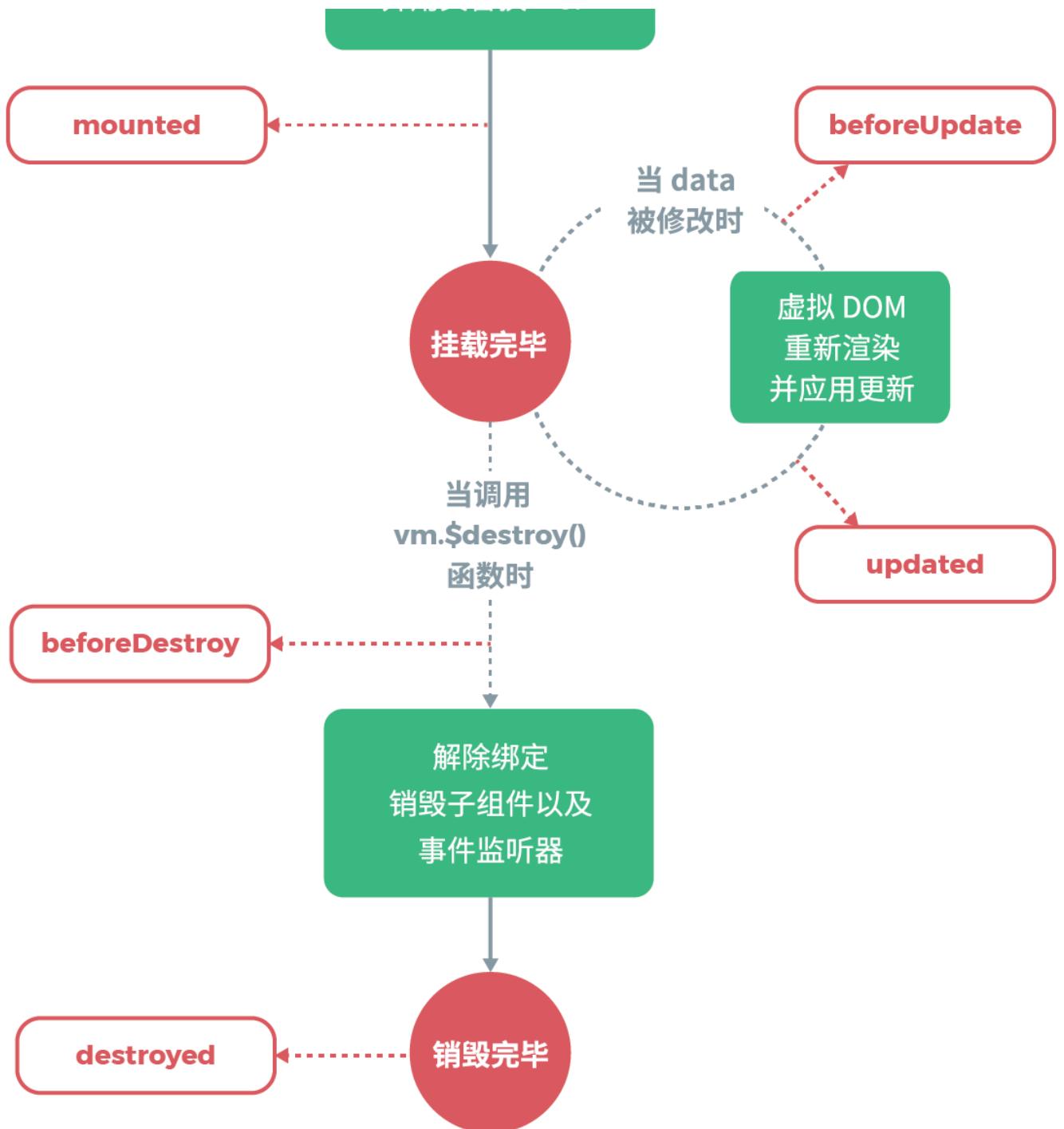




生命周期钩子

所有的生命周期钩子自动绑定 `this` 上下文到实例中，因此你可以访问数据，对属性和方法进行运算





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。

```
<div id="app">
  <div @click="handleClick">点击事件</div>
</div>
```

```
const vm = new Vue({
  el: '#app',
  data: {
    msg: 'hellow world',
  },
  beforeCreate () {
    console.log(this.msg); // undefined
    console.log(this.handleClick); // undefined
    console.log('-----beforeCreate-----');
  },
  methods: {
    handleClick () {
      console.log(handleClick);
    }
  },
  watch: {
    msg: {
      handler () {
        console.log('侦听msg的值');
      },
      immediate: true,
    }
  }
})
```

打印顺序：

```
undefined
undefined
-----beforeCreate-----
侦听msg的值
```

created

在实例创建完成后被立即调用。

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。

如果要在第一时间调用methods中的方法，或者操作data中的数据，可在此钩子中进行操作。

需要注意的是，执行此钩子时，挂载阶段还未开始，\$el 属性目前不可见。

此时，可以进行数据请求，将请求回来的值赋值给data中的数据。

```
<div id="app">
  <div @click="handleClick">点击事件</div>
</div>
```

```

const vm = new Vue({
  el: '#app',
  data: {
    msg: 'hello world',
  },
  created () {
    console.log(this.msg); // hello world
    console.log(this.handleClick); // function () {...}
    console.log(this.$el); // undefined
    console.log('-----created-----');
  },
  methods: {
    handleClick () {
      console.log(handleClick);
    }
  },
  watch: {
    msg: {
      handler () {
        console.log('侦听msg的值');
      },
      immediate: true,
    }
  }
})

```

打印顺序：

```

侦听msg的值
hello world
f handleClick () { console.log(handleClick); }
undefined
-----created-----

```

beforeMount

在挂载开始之前被调用，此时模板已经编译完成，只是未将生成的模板替换el对应的元素。

在此钩子函数中，可以获取到模板最初的状态。

此时，可以拿到vm.\$el，只不过为旧模板

```

const vm = new Vue({
  el: '#app',
  beforeMount () {
    console.log(this.$el);
  }
})

```

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。

在该钩子函数中的vm.\$el为新模板。

执行完该钩子函数后，代表实例已经被完全创建好。

如果要在第一时间，操作页面上的dom节点时，可以在此钩子函数中操作

```

const vm = new Vue({
  el: '#app',
  mounted () {
    console.log(this.$el);
  }
})

```

beforeUpdate

数据更新时调用，发生在虚拟 DOM 打补丁之前。此时数据已经更新，但是DOM还未更新

```
<div id="app">
  {{ msg }}
</div>

const vm = new Vue({
  el: '#app',
  data: {
    msg: 'helllow world',
  },
  beforeUpdate () {
    console.log(this.msg);
    console.log(this.$el);
  },
  methods: {
    handleClick () {
      console.log('handleClick');
    }
  }
})
this.msg = 'xxx';
```

updated

数据更改导致DOM重新渲染后，会执行该钩子函数。

此时数据和dom同步。

beforeDestroy

实例销毁之前调用。在这一步，实例仍然完全可用。

可以在该钩子函数中，清除定时器。

```
<div id="app">
  {{ msg }}
</div>

const vm = new Vue({
  el: '#app',
  data: {
    msg: 'helllow world',
    timer: 0,
  },
  created () {
    this.timer = setInterval(() => {
      console.log('xxx');
    }, 500)
  },
  beforeDestroy () {
    clearInterval(this.timer);
  }
})
```

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。

```
const vm = new Vue({el:'#app'});
//手动销毁vm
vm.$destroy();
```

练习_bilibili首页

```
baseURL: 'https://developer.duyiedu.com/vue/bz'  
url: '/banner'
```

2. 导航

```
baseURL: 'https://developer.duyiedu.com/vue/bz'  
url: '/nav'
```

3. 视频

```
baseURL: 'https://developer.duyiedu.com/vue/bz'
```

```
url: '/video'
```

```
Request:
```

name	type	describe
start	Number	数据起始值
offset	Number	偏移量

组件基础

组件是什么？

组件是可复用的Vue实例，且带有一个名字，例如名字为shanshan-cmp，那么我们则可以在一个通过new Vue创建的根实例中，把这个组件作为自定义元素来使用：

```
<div id="app">  
  <shanshan-cmp></shanshan-cmp>  
</div>  
  
const vm = new Vue({  
  el: '#app'  
})
```

因为组件是可复用的 Vue 实例，所以它们与 new Vue 接收相同的选项，例如 data、computed、watch、methods 以及生命周期钩子等。仅有的例外是像 el 这样根实例特有的选项。

组件注册

全局组件

```
|| Vue.component
```

利用Vue.component创建的组件组件是全局注册的。也就是说它们在注册之后可以用在任何新创建的 Vue 根实例（new Vue）的模板中。

参数：

- {string}
- {Function | Object} [definition]

用法：

注册或获取全局组件。注册还会自动使用给定的id设置组件的名称。

示例：

```
<div id="app">  
  <button-counter></button-counter>  
</div>
```

```

Vue.component('button-counter', {
  data () {
    return {
      count: 0,
    }
  },
  template: `
    <button @click="count ++">你按了我{{ count }}次</button>
  `,
})
const vm = new Vue({
  el: '#app',
})

```

局部组件

在components选项中定义要使用的组件。

对于 components 对象中的每一个属性来说，其属性名就是自定义元素的名字，其属性值就是这个组件的选项对象。优先级高于全局属性

示例：

```

<div id="#app">
  <button-counter></button-counter>
</div>

```

```

const buttonCounter = {
  data () {
    return {
      count: 0
    }
  },
  template: `
    <button @click="count ++">你按了我{{ count }}次</button>
  `,
}
const vm = new Vue({
  el: '#app',
  components: {
    'button-counter': buttonCounter
  }
})

```

组件名

在注册一个组件的时候，我们始终需要给它一个名字。你给予组件的名字可能依赖于你打算拿它来做什么，所以命名要语义化。

组件名大小写

定义组件名的方式有两种：

使用kebab-case（横短线分隔命名）

```
Vue.component('my-component', {/**/});
```

当使用kebab-case定义一个组件时，你必须在引用这个自定义元素时使用kebab-case，例如：`<my-component></my-component>`。

使用PascalCase（大驼峰命名）

```
Vue.component('MyComponent', {/**/});
```

当使用PascalCase定义一个组件时，你在引用这个自定义元素时两种命名法都可以。也就是说`<my-component-name>` 和 `<MyComponentName>` 都是可接受的。注意，尽管如此，直接在 DOM（即字符串模板或单文件组件）中使用时只有 kebab-case 是有效的。

另：我们强烈推荐遵循 W3C 规范中的自定义组件名（字母全小写且必须包含一个连字符）。这会帮助你避免和当前以及未来的 HTML 元素相冲突。

组件复用

可以将组件进行任意次数的复用(自闭合组件不能复用):

```
<div id="#app">
<button-counter></button-counter>
<button-counter></button-counter>
<button-counter></button-counter>
</div>
```

自闭合组件

在单文件组件、字符串模板和 JSX 中没有内容的组件应该是自闭合的—但在 DOM 模板里永远不要这样做。

自闭合组件表示它们不仅没有内容，而且刻意没有内容。其不同之处就好像书上的一页白纸对比贴有“本页有意留白”标签的白纸。而且没有了额外的闭合标签，你的代码也更简洁。

不幸的是，HTML 并不支持自闭合的自定义元素—只有官方的“空”元素。所以上述策略仅适用于进入 DOM 之前 Vue 的模板编译器能够触达的地方，然后再产出符合 DOM 规范的 HTML。

组件的data选项

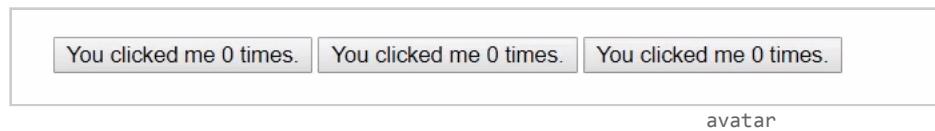
当我们定义一个组件时，它的 data 并不是像这样直接提供一个对象：

```
data: {
  count: 0
}
```

取而代之的是，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对象的独立的拷贝：

```
data () {
  return {
    count: 0
  }
}
```

如果 Vue 没有这条规则，点击一个按钮就可能会像下面一样影响到其它所有实例：



单个根元素

每个组件必须只有一个根元素，当模板的元素大于1时，可以将模板的内容包裹在一个父元素内。

组件_Prop

注册自定义特性

组件默认只是写好结构、样式和行为，使用的数据应由外界传递给组件。

如何传递？注册需要接收的prop，将数据作为一个自定义特性传递给组件。

如：

```
<div id="app">
<video-item
  title="羊村摇"
  poster="https://developer.duyiedu.com/bz/video/955bac93ccb7f240d25a79b2ff6a9fdbda9537bc.jpg@320w_200h.webp"
  play="638000"
  rank="1207"
></video-item>
</div>
```

```
Vue.component('video-item', {
  props: ['title', 'poster', 'play', 'rank'],
})
```

在上述模板中，你会发现我们能够在组件实例中访问这个值，就像访问 `data` 中的值一样：

```
<div id="app">
<video-item
  title="羊村摇"
  poster="https://developer.duyiedu.com/bz/video/955bac93ccb7f240d25a79b2ff6a9fdbda9537bc.jpg@320w_200h.webp"
  play="638000"
  rank="1207"
></video-item>
</div>
```

```
Vue.component('video-item', {
  props: ['title', 'poster', 'play', 'rank'],
  template: `<div>{{ title }}</div>`
})
```

Prop的大小写

HTML 中的特性名是大小写不敏感的，所以浏览器会把所有大写字母解释为小写字母。故：当 传递的prop为 短横线分隔命名时，组件内 的props 应为 驼峰命名 。

如：

```
<div id="app">
  <!-- 在 HTML 中是 kebab-case 的 -->
  <video-item sub-title="hello!"></video-item>
</div>
```

```
Vue.component('video-item', {
  // 在 JavaScript 中是 camelCase 的
  props: ['subTitle'],
  template: '<h3>{{ postTitle }}</h3>'
})
```

要注意的是：如果使用的是字符串模板，那么这个限制就不存在了。

传递静态或动态 Prop

像这样，我们已经知道了可以给 prop 传入一个静态的值：

```
<video-item title="羊村摇"></video-item>
```

若想要传递一个动态的值，可以配合`v-bind`指令进行传递，如：

```
<video-item :title="title"></video-item>
```

传递一个对象的所有属性

如果你想要将一个对象的所有属性都作为 prop 传入，你可以使用不带参数的 `v-bind` 。例如，对于一个给定的对象 `person`：

```
person: {
  name: 'shanshan',
  age: 18
}
```

传递全部属性：

```
<my-component v-bind="person"></my-component>
```

上述代码等价于：

```
<my-component
  :name="person.name"
  :age="person.age"
></my-component>
```

如果想组件内部再套组件，最好用全局组件，局部组件需要在组件内部写components
举例：

```
<div id="app">
  <video-list :list="list"></video-list>
</div>

let videoc = {
  props: ['poster', 'play', 'rank', 'title'],
  template: `<div class="video-item">
    <div class="poster">
      
      <div class="info">
        <div class="play">{{play}}</div>
        <div class="rank">{{rank}}</div>
      </div>
    </div>
    <div class="title">{{title}}</div>
  </div>`;
};
let list = {
  props: ['list'],
  template: `<div class="video-list">
    <videoc v-for="video of list" v-bind="video"></videoc>
  </div>`,
  components: {
    //在这，需要引入组件
    videoc: videoc
  }
};
const vm = new Vue({
  el: '#app',
  data: {
    list: []
  },
  created() {
    axios.get('https://developer.duyiedu.com/vue/bz/video', {
      params: {
        start: 0,
        offset: 12
      }
    }).then(res => {
      this.list = res.data.data;
      console.log(this.list);
    })
  },
  components: {
    //在vim中只用写video-list一个组件
    videoList: list,
  }
})
```

组件_Prop验证

我们可以为组件的 prop 指定验证要求，例如你可以要求一个 prop 的类型为什么。如果说需求没有被满足的话，那么Vue会在浏览器控制台中进行警告，这在开发一个会被别人用到的组件时非常的有帮助。

为了定制 prop 的验证方式，你可以为 props 中的值提供一个带有验证需求的对象，而不是一个字符串数组。例如：

```
Vue.component('my-component', {
  props: {
    title: String,
    likes: Number,
    isPublished: Boolean,
    commentIds: Array,
    author: Object,
    callback: Function,
    contactsPromise: Promise
  }
})
```

上述代码中，对prop进行了基础的类型检查，类型值可以为下列原生构造函数中的一种：String、Number、Boolean、Array、Object、Date、Function、Symbol、任何自定义构造函数、或上述内容组成的数组。需要注意的是 null 和 undefined 会通过任何类型验证。

除基础类型检查外，我们还可以配置高级选项，对prop进行其他验证，如：类型检测、自定义验证和设置默认值。如：

```
Vue.component('my-component', {
  props: {
    title: {
      type: String, // 检查 prop 是否为给定的类型
      default: '杉杉最美', // 为该 prop 指定一个 默认值（在参数list不传时） 对象或数组的默认值必须从一个工厂函数返回，如: default () { return {a: 1, b: 2}; }
      required: true, // 定义该 prop 是否是必填项
      validator (prop) { // 自定义验证函数，该prop的值回作为唯一的参数代入，若函数返回一个falsy的值，那么就代表验证失败
        return prop.length < 140;
      }
    }
  }
})
```

为了更好的团队合作，在提交的代码中，prop 的定义应该尽量详细，至少需要指定其类型。

组件_单向数据流

所有的 prop 都使得其父子 prop 之间形成了一个**单向下行绑定**：父级 prop 的更新(比如说vm的更新)会向下流动到子组件中，但是反过来则不行。这样会防止从子组件意外改变父级组件的状态，从而导致你的应用的数据流向难以理解。

这里有两种常见的试图改变一个 prop 的情形：

1. 这个 prop 用来传递一个初始值；这个子组件接下来希望将其作为一个本地的 prop 数据来使用(如果prop是数组或对象需要深度克隆才可以，因为传递的是引用，所以父组件和子组件数据都会改变)，在后续操作中，会将这个值进行改变。在这种情况下，最好定义一个本地的 data 属性并将这个 prop 用作其初始值：

```
props: ['initialCounter'],
data: function () {
  return {
    //如果是数组或对象需要深度克隆
    counter: this.initialCounter
  }
}
```

2. 这个 prop 以一种原始的值传入且需要进行转换。在这种情况下，最好使用这个 prop 的值来定义一个计算属性：

```
props: ['size'],
computed: {
  normalizedSize: function () {
    return this.size.trim().toLowerCase()
  }
}
```

组件_非Prop特性

非Prop特性指的是，一个未被组件注册的特性。当组件接收了一个非Prop特性时，该特性会被添加到这个组件的根元素上。

替换/合并已有的特性

想象一下 `<my-cmp>` 的模板是这样的：

```
<input type="date" class="b">
```

为了给我们的日期选择器插件定制一个主题，我们可能需要像这样添加一个特别的类名：

```
<my-cmp
  class="my-cmp"
></my-cmp>
```

在这种情况下，我们定义了两个不同的 `class` 的值：

- `my-cmp`, 这是在组件的模板内设置好的
- `b`, 这是从组件的父级传入的

对于绝大多数特性来说，从外部提供给组件的值会替换掉组件内部设置好的值。所以如果传入 `type="text"` 就会替换掉 `type="date"` 并把它破坏！庆幸的是，`class` 和 `style` 特性会稍微智能一些，即两边的值会被合并起来，从而得到最终的值：`my-cmp b`。

禁用特性继承

如果不希望组件的根元素继承特性，那么可以在组件选项中设置 `inheritAttrs: false`。如：

```
Vue.component('my-cmp', {
  inheritAttrs: false,
  // ...
})
```

在这种情况下，非常适合去配合实例的 `$attrs` 属性使用，这个属性是一个对象，键名为传递的特性名，键值为传递特性值。

```
{
  required: true,
  placeholder: 'Enter your username'
}
```

使用 `inheritAttrs: false` 和 `$attrs` 相互配合，我们就可以手动决定这些特性会被赋予哪个元素。如：

```

Vue.component('base-input', {
  //``inheritAttrs: false`` 和 ``$attrs`` 相互配合
  inheritAttrs: false,
  props: ['label', 'value'],
  template: `
    <label>
      {{ label }}
      <input
        v-bind="$attrs"
        v-bind:value="value"
        v-on:input="$emit('input', $event.target.value)"
      >
    </label>
  `,
})

```

举例：把单选框特效赋给input框

```

<div id="app">
  <haha type="radio"></haha>
</div>

```

```

let haha = {
  //``$attrs``中存着{type:'radio'}
  template: `<div><input v-bind="$attrs"></div>`,
  inheritAttrs: false,
  mounted() {
    console.log(this.$attrs);
  }
}
let vm = new Vue({
  el: '#app',
  components: {
    haha: haha
  }
})

```

注意：`inheritAttrs: false` 选项不会影响 `style` 和 `class` 的绑定。

组件_监听组件事件

首先，我们来写一个博文组件，如：

```

Vue.component('blog-post', {
  props: {
    post: {
      type: Object,
    }
  },
  template: `
    <div class="blog-post">
      <h3>{{ post.title }}</h3>
      <button>放大字号</button>
      <div>{{ post.content }}</div>
    </div>
  `,
})

```



```

<div id="app">
  <div :style="{fontSize: postFontSize + 'em'}">
    <blog-post
      v-for="post in posts"
      :key="post.id"
      :post="post"
    >
    </blog-post>
  </div>
</div>

```

```

const vm = new Vue({
  el: '#app',
  data: {
    posts: [
      { title: '标题1', content: '正文内容', id: 0 },
      { title: '标题2', content: '正文内容', id: 1 },
      { title: '标题3', content: '正文内容', id: 2 },
    ],
    postFontSize: 1
  }
})

```

可以看到每一个博文组件中，都有一个按钮，可以去放大页面中字体的字号，也就是说，当点击这个按钮时，我们要告诉父组件改变 `postFontSize` 数据去放大所有博文的文本。碰见这样的情况，该如何做呢？

Vue 实例提供了一个自定义事件来解决这个问题。父组件可以像处理原生DOM元素一样，通过 `v-on` 指令，监听子组件实例的任意事件，如：

```

<div id="app">
  <div :style="{fontSize: postFontSize + 'em'}">
    <blog-post
      ...
      @enlarge-text="postFontSize += 0.1"
    >
    </blog-post>
  </div>
</div>

```

那么，怎么样能够去监听到一个 `enlarge-text` 这么奇怪的事件呢？这就需要在组件内，去主动触发一个**自定义事件**了。

如何触发？

通过调用 `$emit` 方法 并传入事件名称来触发一个事件，如：

```

Vue.component('blog-post', {
  props: {
    ...
  },
  template: `
    <div class="blog-post">
      ...
      <button @click="$emit('enlarge-text')">放大字号</button>
      ...
    </div>
  `,
})

```

这样，父组件就可以接收该事件，更新数据 `pageFontSize` 的值了。

使用事件抛出一个值

在有些情况下，我们可能想让 `<blog-post>` 组件决定它的文本要放大多少。这是可以使用 `$emit` 的第二个参数来提供这个值，如：

```

Vue.component('blog-post', {
  props: {
    ...
  },
  template: `
    <div class="blog-post">
      ...
      <button @click="$emit('enlarge-text', 0.2)">放大字号</button>
      ...
    </div>
  `,
})

```

在父组件监听这个事件时，可以通过 `$event` 访问到被抛出的这个值：

```

<div id="app">
  <div :style="{fontSize: postFontSize + 'em'}">
    <blog-post
      ...
      @enlarge-text="postFontSize += $event"
    >
    </blog-post>
  </div>
</div>

```

或者，将这个事件处理函数写成一个方法：

```

<div id="app">
  <div :style="{fontSize: postFontSize + 'em'}">
    <blog-post
      ...
      @enlarge-text="onEnlargeText"
    >
    </blog-post>
  </div>
</div>

```

那么，这个值将会作为第一个参数，传入这个方法：

```

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

```

举例：使用\$emit

```

<!-- 使用$emit -->
<zujian @shijian-c="countjia" :num='count'></zujian>

//使用$emit
Vue.component('zujian', {
  props: {
    'num': Number
  },
  template: `<div><button @click='$emit("shijian-c",2)'>{{num}}</button></div>`,
});
//vm中
methods: {
  countjia(event) {
    this.count += event;
  }
}

```

事件名

不同于组件和prop，事件名不存在任何自动化的大小写转换。而是触发的事件名需要完全匹配监听这个事件所有的名称。如果触发一个camelCase名字的事件：

```
this.$emit('myEvent')
```

则监听这个名字的kebab-case版本是不会有任何效果的。

```
<!-- 没有效果 -->
<my-component v-on:my-event="doSomething"></my-component>
```

与组件和prop不同的是，事件名不会被当作一个 JS 变量名或者属性名，所以就没有理由使用camelCase 或 PascalCase 了。

并且 v-on 事件监听器在 DOM 模板中会被自动转换为全小写，所以 @myEvent 将会变成 @myevent，导致 myEvent 不可能被监听到。

因此，推荐始终使用 kebab-case 的事件名。

将原生事件绑定到组件

在组件上去监听事件时，我们监听的是组件的自动触发的自定义事件，但是在一些情况下，我们可能想要在一个组件的根元素上直接监听一个原生事件。这是，可以使用 `v-on` 指令的 `.native` 修饰符，如：

```
<base-input @focus.native="onFocus" @blur.native="onBlur"></base-input>
```

```
Vue.component('base-input', {
  template: `
    <input type="text" />
  `,
})
```

这样处理，在有些时候是很有用的，不过在尝试监听一个类似 `<input>` 元素时，这并不是一个好主意，例如 `<base-input>` 组件可能做了重构，如：

```
<label>
  姓名:
  <input type="text">
</label>
```

可以看到，此时组件的根元素实际上是一个元素，那么父级的 `.native` 监听器将静默失败。它不会产生任何报错，但是 `onFocus` 处理函数不会如预期被调用。

为了解决这个问题，Vue 提供了一个 `$listeners` 属性，它是一个对象，里面包含了作用在这个组件上的所有监听器。例如：

```
{
  focus: function (event) { /* ... */ },
  blur: function (event) { /* ... */ },
}
```

有了这个 `$listeners` 属性，我们可以配合 `v-on="$listeners"` 将所有的事件监听器指向这个组件的某个特定的子元素，如：

```
Vue.component('base-input', {
  template: `
    <label>
      姓名:
      <input v-on="$listeners" />
    </label>
  `,
})
```

举例：使用 `$listeners`

```
<z-listener @focus="onfocus" @blur="onblur"></z-listener>

// 使用 listener
Vue.component('z-listener', {
  mounted() {
    console.log(this.$listeners);
  },
  template: `<label>姓名: <input v-on="$listeners"></label>`
})
```

在组件上使用 `v-model`

由于自定义事件的出现，在组件上也可以使用 `v-model` 指令。

在 `input` 元素上使用 `v-model` 指令时，相当于绑定了 `value` 特性以及监听了 `input` 事件：

```
<input v-model="searchText" />
```

等价于：

```
<input  
  :value="searchText"  
  @input="searchText = $event.target.value"  
>
```

当把v-model指令用在组件上时：

```
<base-input v-model="searchText" />
```

则等价于：

```
<base-input  
  :value="searchText"  
  @input="searchText = $event"  
>
```

同 input 元素一样，在组件上使用v-model指令，也是绑定了value特性，监听了input事件。

所以，为了让 v-model 指令正常工作，这个组件内的 <input> 必须：

- 将其value特性绑定到一个叫 value 的prop 上
 - 在其input事件被触发时，将新的值通过自定义的input事件抛出
- 如：

```
Vue.component('base-input', {  
  props: ['value'],  
  template: `  
    <input  
      :value="value"  
      @input="$emit('input', $event.target.value)"  
    />  
  `,  
})
```

这样操作后，v-model就可以在这个组件上工作起来了。

举例：使用v-model

```
<z-model v-model="ztext"></z-model>
```

```
//使用v-model 因为v-model是语法糖 所以props 中必须有value  
Vue.component('z-model', {  
  props: ['value'],  
  template: `<input :value="value" @input="$emit('input', $event.target.value)">`  
})
```

举例：不使用v-model 实现双向绑定

```
<!-- 不使用v-model 实现双向绑定 -->  
<un-model :zvalue='ztext' @zinput="ztext = $event"></un-model>
```

```
//不使用v-model 实现双向绑定  
Vue.component('un-model', {  
  props: ['zvalue'],  
  template: `<input :value='zvalue' @input="$emit('zinput',$event.target.value)" />`  
})
```

通过上面的学习，我们知道了，一个组件上的 v-model 默认会利用名为 value 的 prop 和名为 input 的事件，但是像单选框、复选框等类型的输入控件可能会将 value 特性用于不同的目的。碰到这样的情况，我们可以利用 model 选项来避免冲突：

```

Vue.component('base-checkbox', {
  model: {
    prop: 'checked',
    event: 'change'
  },
  props: {
    checked: Boolean
  },
  template: `
    <input
      type="checkbox"
      :checked="checked"
      @change="$emit('change', $event.target.checked)"
    >
  `
})

```

使用组件:

```
<base-checkbox v-model="lovingVue"></base-checkbox>
```

这里的 lovingVue 的值将会传入这个名为 checked 的 prop。同时当 触发一个 change 事件并附带一个新的值的时候，这个 lovingVue 的属性将会被更新。

.sync 修饰符

除了使用 v-model 指令实现组件与外部数据的双向绑定外，我们还可以用 v-bind 指令的修饰符 .sync 来实现。

那么，该如何实现呢？

先回忆一下，不利用 v-model 指令来实现组件的双向数据绑定：

```

<base-input :value="searchText" @input="searchText = $event"></base-input>

Vue.component('base-input', {
  props: ['value'],
  template: `
    <input
      :value="value"
      @input="$emit('input', $event.target.value)"
    >
  `
})

```

那么，我们也可以试着，将监听的事件名进行更改，如：

```

<base-input :value="searchText" @update:value="searchText = $event"></base-input>

Vue.component('base-input', {
  props: ['value'],
  template: `
    <input
      :value="value"
      @input="$emit('update:value', $event.target.value)"
    >
  `
})

```

这样也是可以实现双向数据绑定的，那么和 .sync 修饰符 有什么关系呢？

此时，我们对代码进行修改：

```
<base-input :value.sync="searchText"></base-input>
```

```

Vue.component('base-input', {
  props: ['value'],
  template: `
    <input
      :value="value"
      @input="$emit('update:value', $event.target.value)"
    />
  `)
})

```

所以，`.sync` 修饰符 本质上也是一个语法糖，在组件上使用：

```
<base-input :value.sync="searchText"></base-input>
```

等价于：

```

<base-input
  :value="searchText"
  @update:value="searchText = $event"
/>

```

举例：使用`.sync`

```

<z-sync :value.sync='zsynctext'></z-sync>

//使用sync 实现双向绑定 props中value, $emit中的update:value需固定
Vue.component('z-sync', {
  props: ['value'],
  template: `<input :value='value' @input="$emit('update:value',$event.target.value)" />`
})

```

当我们用一个对象同时设置多个prop时，也可以将`.sync`修饰符和`v-bind`配合使用：

```
<base-input v-bind.sync="obj"></base-input>
```

相当于

```
<base-input :a.sync="value1" :b.sync="value2" :c.sync="value2" :d.sync="value2"></base-input>
```

注意：

- 带有`.sync`修饰符的`v-bind`指令，只能提供想要绑定的属性名，**不能**和表达式一起使用，如：`:title.sync="1+1"`，这样操作是无效的
- 将`v-bind.sync`用在一个字面量对象上，如`v-bind.sync="{ title: 'haha' }"`，是无法工作的，因为在解析一个像这样的复杂表达式的时候，有很多边缘情况需要考虑。

v-model VS .sync

先明确一件事情，在`vue 1.x`时，就已经支持`.sync`语法，但是此时的`.sync`可以完全在子组件中修改父组件的状态，造成整个状态的变换很难追溯，所以官方在`2.0`时移除了这个特性。然后在`vue2.3`时，`.sync`又回归了，跟以往不同的是，现在的`.sync`完完全全就是一个语法糖的作用，跟`v-model`的实现原理是一样的，也不容易破坏原有的数据模型，所以使用上更安全也更方便。

- 两者都是用于实现双向数据传递的，实现方式都是语法糖，最终通过`prop + 事件`来达成目的。
- `vue 1.x`的`.sync`和`v-model`是完全两个东西，`vue 2.3`之后可以理解为一类特性，使用场景略微有区别
- 当一个组件对外只暴露一个受控的状态，切都符合统一标准的时候，我们会使用`v-model`来处理。`.sync`则更为灵活，凡是需要双向数据传递时，都可以去使用。

组件_插槽

和`HTML`元素一样，我们经常需要向一个组件传递内容，像这样：

```
<my-cmp>
  Something bad happened.
</my-cmp>
```

如果有这样的需求，我们就可以通过插槽来做。

插槽内容

通过插槽，我们可以这样合成组件：

```
<my-cmp>
  写在组件标签结构中的内容
</my-cmp>
```

组件模板中可以写成：

```
<div>
  <slot></slot>
</div>
```

等于

```
<div>
  写在组件标签结构中的内容
</div>
```

当组件渲染时，`<slot></slot>` 将会被替换为“写在组件标签结构中的内容”。

插槽内可以包含任何模板代码，包括HTML和其他组件。

如果 `<my-cmp>` 没有包含 `<slot>` 元素，则该组件起始标签和结束标签之间的任何内容都会被抛弃。

编译作用域

当在插槽中使用数据时：

```
<my-cmp>
  这是插槽中使用的数据: {{ user }}
</my-cmp>
```

该插槽跟模板的其他地方一样可以访问相同的实例属性，也就是相同的“作用域”，而不能访问 `<my-cmp>` 的作用域。

请记住：

父级模板里的所有内容都是在父级作用域中编译的；子模板里的所有内容都是在子作用域中编译的。

后备内容

我们可以设置默认插槽，它会在没有提供内容时被渲染，如，在 `<my-cmp>` 组件中：

```
Vue.component('my-cmp', {
  template: `
    <button type="submit">
      <slot></slot>
    </button>
  `
})
```

我们希望这个 `<button>` 内绝大多数情况下都渲染文本“Submit”，此时就可以将“Submit”作为后备内容，如：

```
Vue.component('my-cmp', {
  template: `
    <button type="submit">
      <slot>Submit</slot>
    </button>
  `
})
```

当使用组件未提供插槽时，后备内容将会被渲染。

```
<my-cmp><my-cmp>
```

如果提供插槽，则后备内容将会被取代。

```
<my-cmp>有我，就不会显示slot中后备内容</my-cmp>
```

具名插槽

有时我们需要多个插槽，如 `<my-cmp>` 组件：

```
Vue.component('my-cmp', {
  template: `
    <div class="container">
      <header>
        <!-- 页头 -->
      </header>
      <main>
        <!-- 主要内容 -->
      </main>
      <footer>
        <!-- 页脚 -->
      </footer>
    </div>
  `
})
```

此时，可以在 `<slot>` 元素上使用一个特殊的特性：`name`。利用这个特性定义额外的插槽：

```
Vue.component('my-cmp', {
  template: `
    <div class="container">
      <header>
        <slot name="header"></slot>
      </header>
      <main>
        <slot></slot>
      </main>
      <footer>
        <slot name="footer"></slot>
      </footer>
    </div>
  `
})
```

一个不带 `name` 的 `<slot>` 出口会带有隐含的名字“`default`”。

在向具名插槽提供内容的时候，我们可以在一个 `<template>` 元素上使用 `v-slot` 指令，并以 `v-slot` 的参数的形式提供其名称：

```

<my-cmp>
<template v-slot:header>
  <h1>头部</h1>
</template>

<p>内容</p>
<p>内容</p>

<template v-slot:footer>
  <p>底部</p>
</template>
</my-cmp>

```

现在 `<template>` 元素中的所有内容都会被传入相应的插槽。任何没有被包裹在带有 `v-slot` 的 `<template>` 中的内容都会被视为默认插槽的内容。为了模板更清晰，也可以写成以下这样：

```

<my-cmp>
<template v-slot:header>
  <h1>头部</h1>
</template>

<template v-slot:default>
  <p>内容</p>
  <p>内容</p>
</template>

<template v-slot:footer>
  <p>底部</p>
</template>
</my-cmp>

```

注意：`v-slot` 只能添加在 `<template>` 上，只有一种例外情况。

作用域插槽

为了能够让插槽内容访问子组件的数据，我们可以将子组件的数据作为 `<slot>` 元素的一个特性绑定上去：

```

Vue.component('my-cmp', {
  data () {
    return {
      user: {
        name: '杉杉',
        age: 18,
      }
    }
  },
  template: `
    <span>
      <slot v-bind:user="user"></slot>
    </span>
  `,
})

```

绑定在 `<slot>` 元素上的特性被称为 **插槽 prop**。

那么在父级作用域中，我们可以给 `v-slot` 带一个值来定义我们提供的插槽 `prop` 的名字：

```

<div id="app">
<my-cmp>
  <template v-slot:default="slotProps">
    {{ slotProps.user.name }}
  </template>
</my-cmp>
</div>

```

独占默认插槽的缩写语法（不能跟具名插槽同时使用）

当被提供的内容只有默认插槽时，组件的标签可以被当作插槽的模板来使用，此时，可以将 `v-slot` 直接用在组件上：

```
<my-cmp v-slot:default="slotProps">
  {{ slotProps.user.name }}
</my-cmp>
```

也可以更简单：

```
<my-cmp v-slot="slotProps">
  {{ slotProps.user.name }}
</my-cmp>
```

注意：默认插槽的缩写语法不能和具名插槽混用，因为它会导致作用域不明确。

```
<!-- 无效，会导致警告 -->
<my-cmp v-slot="slotProps">
  {{ slotProps.user.name }}
  <template v-slot:other="otherSlotProps">
    slotProps 在这里是不合法的
  </template>
</my-cmp>
```

只要出现多个插槽，就需要为所有的插槽使用完整的基于 `<template>` 的语法。

解构插槽Prop

我们可以使用解构传入具体的插槽prop，如：

```
<my-cmp v-slot="{ user }">
  {{ user.name }}
</my-cmp>
```

这样模板会更简洁，尤其是在为插槽提供了多个prop时。

此外还可以有其他可能，如prop重命名：

```
<my-cmp v-slot="{ user: person }">
  {{ person.name }}
</my-cmp>
```

以及自定义后备内容，当插槽prop是`undefined`时生效：

```
<my-cmp v-slot="{ user = { name: 'Guest' } }">
  {{ user.name }}
</my-cmp>
```

动态插槽名

Vue 2.6.0新增

```
<my-cmp>
  <template v-slot:[dynamicSlotName]>
    ...
  </template>
</my-cmp>
```

举例：

```
<!-- 动态插槽 -->
<slot-dt>
  <template v-slot:[dongtai]>
    我是动态的
  </template>
</slot-dt>
```

```
Vue.component('slot-dt', {
  template: `<div><slot name="header"></slot></div>`
})
const vm = new Vue({
  el: '#app',
  data: {
    dongtai: 'header'
  }
})
```

具名插槽的缩写

|| Vue 2.6.0新增

跟 v-on 和 v-bind 一样， v-slot 也有缩写，将 v-slot: 替换为 # 。

```
<my-cmp>
<template #header>
  <h1>头部</h1>
</template>

<template #default>
  <p>内容</p>
  <p>内容</p>
</template>

<template #footer>
  <p>底部</p>
</template>
</my-cmp>
```

当然，和其它指令一样，该缩写只在其有参数的时候才可用。

废弃了的语法

带有 slot 特性的具名插槽

|| 自 2.6.0 起被废弃

```
<my-cmp>
<template slot="header">
  <h1>头部</h1>
</template>

<template>
  <p>内容</p>
  <p>内容</p>
</template>

<template slot="footer">
  <p>底部</p>
</template>
</my-cmp>
```

带有 slot-scope 特性的作用域插槽

|| 自 2.6.0 起被废弃

```
<my-cmp>
<template slot="default" slot-scope="slotProps">
  {{ slotProps.user.name }}
</template>
</my-cmp>
```

组件_动态组件

基本使用

当我们在一个多标签的界面中，在不同组件之间进行动态切换是非常有用的。

```
<div id="app">
  <button
    v-for="page in pages"
    @click="pageCmp = page.cmp"
    :key="page.id"
  >{{ page.name }}</button>
  <component :is="pageCmp"></component>
</div>

Vue.component('base-post', {
  data () {
    return {
      postCmp: '',
      posts: [
        { title: "标题1", content: { template: `<div>内容1</div>`}, id: 11},
        { title: "标题2", content: { template: `<div>内容2</div>`}, id: 12},
        { title: "标题3", content: { template: `<div>内容3</div>`}, id: 13},
      ],
    }
  },
  mounted () {
    this.postCmp = this.posts[0].content;
  },
  template: `
    <div>
      <button
        v-for="post in posts"
        @click="postCmp = post.content"
        :key="post.id"
      >{{ post.title }}</button>
      <component :is="postCmp"></component>
    </div>
  `
})
Vue.component('base-more', {
  template: `<div>更多内容</div>`
})

const vm = new Vue({
  el: '#app',
  data: {
    pages: [
      { name: '博客', cmp: 'base-post', id: 0},
      { name: '更多', cmp: 'base-more', id: 1}
    ],
    pageCmp: 'base-post'
  }
})
```

通过上面方法，我们可以实现组件间的切换，能够注意到的是：每一次切换标签时，都会创建一个新的组件实例，重新创建动态组件在更多情况下是非常有用的，但是在这个案例中，我们会更希望哪些标签的组件实例能够在它们第一次被创建的时候缓存下来。为了解决这个问题，我们可以用一个 `<keep-alive>` 元素将动态组件包裹起来。如：

```
<!-- 失活的组件将会被缓存! -->
<keep-alive>
  <component v-bind:is="pageCmp"></component>
</keep-alive>
```

注意：`<keep-alive>` 要求被切换到的组件都有自己的名字，不论是通过组件的 `name` 选项还是局部/全局注册。

keep-alive

`<keep-alive>` 包裹动态组件时，会缓存不活动的组件实例，而不是销毁它们。`<keep-alive>` 是一个抽象组件：它自身不会渲染一个 DOM 元素，也不会出现在父组件链中。

当组件在 `<keep-alive>` 内被切换，它的 `activated` 和 `deactivated` 这两个生命周期钩子函数将会被对应执行。

activated & deactivated

activated: keep-alive 组件激活时调用。
deactivated: keep-alive 组件停用时调用。

组件_处理边界情况

接下来我们要学习的都是和处理边界情况有关的功能，即一些需要对 Vue 的规则做一些小调整的特殊情况。需要注意的是，这些功能都是有劣势或危险场景的。

访问元素 & 组件

在绝大多数情况下，我们最好不要触达另一个组件实例内部或手动操作 DOM 元素。不过也确实在一些情况下做这些事情是合适的。

访问根实例

在每个子组件中，可以通过 \$root 访问根实例。

```
// Vue 根实例
new Vue({
  data: {
    foo: 1
  },
  computed: {
    bar () { /* ... */ }
  },
  methods: {
    baz () { /* ... */ }
  }
})
```

所有的子组件都可以将这个实例作为一个全局 store 来访问或使用。

```
// 获取根组件的数据
this.$root.foo

// 写入根组件的数据
this.$root.foo = 2

// 访问根组件的计算属性
this.$root.bar

// 调用根组件的方法
this.$root.baz()
```

在demo或在有少量组件的小型应用中使用是非常方便的。但是在大型应用里使用就会很复杂了。所以，我们还是要用Vuex（后面会学）来管理应用的状态。

举例：使用this.\$root

```
<div id="app">
  <my-cmp>
    <my-cmp1>
      <my-cmp2></my-cmp2>
    </my-cmp1>
  </my-cmp>
</div>
```

```

Vue.component('my-cmp', {
  template: `<div>1<slot></slot></div>`,
  created() {
    console.log(this.$root.name);
  }
})
Vue.component('my-cmp1', {
  template: `<div>2<slot></slot></div>`,
  created() {
    console.log(this.$root.name);
  }
})
Vue.component('my-cmp2', {
  template: `<div>3</div>`,
  created() {
    console.log(this.$root.name);
  }
})
const vm = new Vue({
  el: '#app',
  data: {
    name: 'mao~'
  },
  created() {
    console.log(this.$root.name);
  }
})

```

访问父级组件实例

在子组件中，可以通过 `$parent` 访问 父组件实例。这可以替代将数据以`prop`的方式传入子组件的方式。

如：

```

<cmp-parent>
  <cmp-a></cmp-a>
</cmp-parent>

```

若 `cmp-parent` 需要共享一个属性 `share`，它的所有子元素都需要访问 `share` 属性，在这种情况下 `cmp-a` 可以通过 `this.$parent.share` 的方式访问 `share`。

但是，通过这种模式构建出来的组件内部仍然容易出现问题。比如，我们在`cmp-a` 中嵌套一个一个子组件 `cmp-b`，如：

```

<cmp-parent>
  <cmp-a>
    <cmp-b></cmp-b>
  </cmp-a>
</cmp-parent>

```

那么，在`cmp-b`组件中去访问`share`时，需要先查看一下，其父组件中是否存在`share`，如果不存在，则在向上一级查找，落实到代码上为：

```
var share = this.$parent.share || this.$parent.$parent.share;
```

举例：`this.$parent`

```

<div id="app">
  <my-cmp>
    <my-cmp1>
      <my-cmp2></my-cmp2>
    </my-cmp1>
  </my-cmp>
</div>

```

```

Vue.component('my-cmp', {
  data() {
    return {
      share: 'share'
    }
  },
  template: `<div>1<slot></slot></div>`,
  created() {
    console.log(this.share);
  }
})
Vue.component('my-cmp1', {
  template: `<div>2<slot></slot></div>`,
  created() {
    console.log(this.$parent.share);
  }
})
Vue.component('my-cmp2', {
  template: `<div>3</div>`,
  created() {
    console.log(this.$parent.$parent.share);
  }
})

```

这样做，很快组件就会失控：触达父级组件会使应用更难调试和理解，尤其是当变更父组件数据时，过一段时间后，很难找出变更是从哪里发起的。

碰到上述情况，可以使用依赖注入解决。

依赖注入

在上面的例子中，利用 `$parent` 属性，没有办法很好的扩展到更深层级的嵌套组件上。这也是依赖注入的用武之地，它用到了两个新的实例选项：`provide` 和 `inject`。

这两个实例方法不能在 `vm = new Vue(...)` 里使用。只能在组件中使用。

`provide` 选项允许我们指定想要提供给后代组件的数据/方法，例如：

```

Vue.component('cmp-parent', {
  provide () {
    return {
      share: this.share,
    }
  },
  data () {
    return {
      share: 'share',
    }
  },
  template: `<div>cmp-parent</div>`
})

```

然后再任何后代组件中，我们都可以使用 `inject` 选项来接受指定想要添加在实例上的属性。

```

Vue.component('cmp-a', {
  inject: ['share'],
  template: `<div>cmp-a</div>`
})

```

举例：`provide` `inject` 的使用

```

Vue.component('my-cmp', {
  provide() {
    return {
      share: this.share
    }
  },
  data() {
    return {
      share: 'share'
    }
  },
  template: `<div>1<slot></slot></div>`,
  created() {
    console.log(this.share);
  }
})
Vue.component('my-cmp1', {
  template: `<div>2<slot></slot></div>`,
  inject: ['share'],
  created() {
    console.log(this.share);
  }
})

```

相比 \$parent 来说，这个用法可以让我们在任意后代组件中访问share，而不需要暴露整个 cmp-parent 实例。这允许我们更好的持续研发该组件，而不需要担心我们可能会改变/移除一些子组件依赖的东西。同时这些组件之间的接口是始终明确定义的，就和 props 一样。

实际上，你可以把依赖注入看作一部分“大范围有效的 prop”，除了：

- 祖先组件不需要知道哪些后代组件使用它提供的属性
- 后代组件不需要知道被注入的属性来自哪里

然而，依赖注入还是有负面影响的。它将你应用程序中的组件与它们当前的组织方式耦合起来，使重构变得更加困难。同时所提供的属性是非响应式的。这是出于设计的考虑，因为使用它们来创建一个中心化规模化的数据跟使用 \$root 做这件事都是不够好的。如果你想要共享的这个属性是你的应用特有的，而不是通用化的，或者如果你想在祖先组件中更新所提供的数据，那么这意味着你可能需要换用一个像 Vuex 这样真正的状态管理方案了。

访问子组件实例或子元素

尽管存在prop和事件，但是有时候我们仍可能需要在JS里直接访问一个子组件，那么此时，我们可以通过 ref 特性为子组件赋予一个ID引用：

```
<my-cmp ref="cmp"></my-cmp>
```

这样就可以通过this.\$refs.cmp 来访问 <my-cmp> 实例。

注意：

1. 在vm(Vue实例)中可以看所有内部组件的this.refs.xxx，组件只能看自己内部的this.refs.xxx

2. 如果把 ref='xxx' 写在了父组件的子组件中，父组件需要加 <slot></slot>

ref 也可以 对指定DOM元素进行访问，如：

```
<input ref="input" />
```

那么，我们可以通过 this.refs.input 来访问到该DOM元素。举例：this.\$refs.xxx的使用

```

<div id="app">
  <my-cmp ref="cmp">
    <my-cmp1 ref="cmp1"></my-cmp1>
  </my-cmp>
  <input v-for="item in 3" ref="input" />
</div>

```

```

Vue.component('my-cmp', {
  data() {
    return {
      share: 'share'
    }
  },
  //如果不加slot 父组件中的cmp1则不会被访问到
  template: `<div class="one"><slot></slot></div>`,
})
Vue.component('my-cmp1', {
  template: `<div class="two"></div>`
})
const vm = new Vue({
  el: '#app',
  data: {
    name: 'mao~'
  },
  mounted() {
    console.log(this.$refs.input);
    console.log(this.$refs.cmp1);
  }
})

```

当`ref` 和 `v-for` 一起使用时，得到的引用将会是一个包含了对应数据源的这些子组件的数组。

注意：`$refs` 只会在组件渲染完成之后生效，并且它们不是响应式的。应该避免在模板或计算属性中访问 `$refs`。

程序化的事件监听器

除了 `v-on` 和 `$emit` 外，`Vue` 实例在其事件接口中还提供了其它的方法。我们可以：

- 通过 `$on(eventName, eventHandler)` 监听一个事件
- 通过 `$once(eventName, eventHandler)` 一次性监听一个事件
- 通过 `$off(eventName, eventHandler)` 停止监听一个事件

这几个方法一般不会被用到，但是，当需要在一个组件实例上手动监听事件时，他们是可以派的上用场的。

例如，有时我们会在组件中集成第三方库：

```

Vue.component('my-cmp', {
  // 一次性将这个日期选择器附加到一个输入框上
  // 它会被挂载到 DOM 上。
  mounted () {
    // Pikaday 是一个第三方日期选择器的库
    this.picker = new Pikaday({
      field: this.$refs.input,
      format: 'YYYY-MM-DD',
    })
  },
  // 在组件被销毁之前,
  // 也销毁这个日期选择器。
  beforeDestroy () {
    this.picked.destroy();
  },
  template: `
    <div>
      <input type="text" ref="input" />
      <button @click="$destroy()">销毁组件</button>
    </div>
  `,
})

```

使用上面的方法，有两个潜在的问题：

- 它需要在这个组件实例中保存这个 `picker`，如果可以的话最好只有生命周期钩子可以访问到它。这并不算严重的问题，但是它可以被视为杂物。
- 我们的建立代码独立于我们的清理代码，这使得我们比较难于程序化地清理我们建立的所有东西。

所有，我们可以通程序化的监听器解决这两个问题：

```

Vue.component('my-cmp', {
  mounted () {
    var picker = new Pikaday({
      field: this.$refs.input,
      format: 'YYYY-MM-DD',
    })
    this.$once('hook:beforeDestroy', () => {
      picker.destroy();
    })
  },
  template: `
    <div>
      <input type="text" ref="input" />
      <button @click="$destroy()">销毁组件</button>
    </div>
  `
})

```

使用了这个策略，我们还可以让多个输入框元素使用不同的pikaday：

```

Vue.component('my-cmp', {
  mounted () {
    this.datePicker('inputA');
    this.datePicker('inputB');
  },
  methods: {
    datePicker (refName) {
      var picker = new Pikaday({
        field: this.$refs[refName],
        format: 'YYYY-MM-DD',
      })
      this.$once('hook:beforeDestroy', () => {
        picker.destroy();
      })
    },
  },
  template: `
    <div>
      <input type="text" ref="inputA" />
      <input type="text" ref="inputB" />
      <button @click="$destroy()">销毁组件</button>
    </div>
  `
})

```

注意，即便如此，如果你发现自己不得不在单个组件里做很多建立和清理的工作，最好的方式通常还是创建更多的模块化组件，在这个例子中，我们推荐创建一个可复用的 `<input-datepicker>` 组件。

循环引用

递归组件

组件是可以在它们自己的模板中调用自身的，不过它们只能通过`name`选项来做这件事：

```
name: 'my-cmp'
```

不过当使用 `Vue.component` 全局注册一个组件时，全局的ID会自动设置为该组件的 `name` 选项。

```
Vue.component('my-cmp', { /* */});
```

稍有不慎，递归组件就可能导致无限循环：

```
name: 'my-cmp',
template: `<div><my-cmp /></div>`
```

类似上述的组件将会导致“`max stack size exceeded`”错误，所以要确保递归调用是条件性的（例如使用一个最终会得到 `false` 的 `v-if`）。

组件之间的循环引用

有时，在去构建一些组件时，会出现组件互为对方的后代/祖先：

```
Vue.component('cmp-a', {
  template: `
    <div>
      <cmp-b></cmp-b>
    </div>
  `
})

Vue.component('cmp-b', {
  template: `
    <div>
      <cmp-a></cmp-a>
    </div>
  `
})
```

此时，我们使用的是全局注册组件，并不会出现悖论，但是如果使用的为局部组件就会出现悖论。

但是即使用了全局注册组件，在使用webpack去导入组件时，也会出现一个错误： Failed to mount component: template or render function not defined。

模块系统发现它需要 A，但是首先 A 依赖 B，但是 B 又依赖 A，但是 A 又依赖 B，如此往复。这变成了一个循环，不知道如何不经过其中一个组件而完全解析出另一个组件。为了解决这个问题，我们需要给模块系统一个点：“A 反正是需要 B 的，但是我们不需要先解析 B。”

```
beforeCreate () {
  this.$options.components.CmpB = require('./tree-folder-contents.vue').default;
}
```

或者，在本地注册组件的时候，你可以使用 webpack 的异步 import：

```
components: {
  CmpB: () => import('./tree-folder-contents.vue')
}
```

模板定义的替代品

内联模板

在使用组件时，写上特殊的特性： inline-template，就可以直接将里面的内容作为模板而不是被分发的内容（插槽）。

```
<my-cmp inline-template>
  <div>
    <p>These are compiled as the component's own template.</p>
    <p>Not parent's transclusion content.</p>
  </div>
</my-cmp>
```

不过， inline-template 会让模板的作用域变得更加难以理解。所以作为最佳实践，请在组件内优先选择 template 选项或 .vue 文件里的一个 <template> 元素来定义模板。

X-Template

另一个定义模板的方式是在一个 <script> 元素中，并为其带上 text/x-template 的类型，然后通过一个 id 将模板引用过去。例如：

```
<script
  type="text/x-template"
  id="hello-world-template">
  <p>Hello hello hello</p>
</script>
```

```
Vue.component('hello-world', {
  template: '#hello-world-template'
})
```

这些可以用于模板特别大的 demo 或极小型的应用，但是其它情况下请避免使用，因为这会将模板和该组件的其它定义分离开。

控制更新

强制更新

当更改了某个数据，页面未重新渲染时，可以调用 `$forceUpdate` 来做一次强制更新。

但是在做强制更新前，需要留意数组或对象的变更检测注意事项，99.9%的情况，都是在某个地方做错了事，如果做了上述检查，仍未发现问题，那么可以通过 `$forceUpdate` 来更新。

通过v-once创建低开销的静态组件

渲染普通的 HTML 元素在 Vue 中是非常快速的，但有的时候你可能有一个组件，这个组件包含了大量静态内容。在这种情况下，你可以在根元素上添加 `v-once` 特性以确保这些内容只计算一次然后缓存起来，就像这样：

```
Vue.component('terms-of-service', {
  template: `
    <div v-once>
      <h1>Terms of Service</h1>
      ... a lot of static content ...
    </div>
  `
})
```

试着不要过度使用这个模式。当你需要渲染大量静态内容时，极少数的情况下它会给你带来便利，除非你非常留意渲染变慢了，不然它完全是没有必要的—再加上它在后期会带来很多困惑。例如，设想另一个开发者并不熟悉 `v-once` 或漏看了它在模板中，他们可能会花很多个小时去找出模板为什么无法正确更新。

组件_通信

props (推荐)

父组件传递数据给子组件时，可以通过特性传递。

推荐使用这种方式进行父->子通信。

举例：

```
<prop-cmp :ceshi="father"></prop-cmp>

//props 父组件传递数据给子组件 推荐
Vue.component('prop-cmp', {
  props: ['ceshi'],
  template: `
    <div>测试prop{{ceshi}}</div>
  `
})
const vm = new Vue({
  el: '#app',
  data: {
    father: '父组件'
  }
});
```

\$emit (推荐)

子组件传递数据给父组件时，触发事件，从而抛出数据。

推荐使用这种方式进行子->父通信。

举例：

```
<emit-cmp @ent="myEvent"></emit-cmp>

Vue.component('emit-cmp', {
  template: `
    <button @click="$emit('ent','emit')">emit</button>
  `
})
const vm = new Vue({
  el: '#app',
  methods: {
    myEvent(child) {
      console.log(child);
    }
  }
});
```

v-model 双向绑定

举例：

我们可以在后台使用vm.ztext = 'xxx'给input框赋值，也可以通过vm.ztext查询input框里的值

```
<model-cmp v-model="ztext"></model-cmp>

Vue.component('model-cmp', {
  //props 里必须有value
  props: ['value'],
  template: `
    <input :value="value" @input="$emit('input',$event.target.value)" />
  `
})
const vm = new Vue({
  el: '#app',
  data: {
    //model
    ztext: ''
  }
});
```

注意：

关于v-model执行方法，可在被调用子组件中添加选项

```
model: {
  //model 为 v-model使用
  //变量名
  prop: "dateP",
  //事件名 自己定义事件名
  event: "date-choose",
},
```

举例：v-model自定义事件和变量的使用

父组件

```
<data-picker v-model="dateP"></data-picker>
```

```

import datePicker from "./DatePicker.vue";
export default {
  components: {
    DatePicker: datePicker,
  },
  data() {
    return {
      //这时 dateP为传给子组件的值
      dateP: new Date(),
    };
  }
};

```

子组件

```

export default {
  props: {
    //这个dateP就是父组件传来的值
    dateP: {
      type: Date,
      //默认值为数据或对象时 用函数返回
      //default: () => new Date(),
    },
  },
  model: {
    //model 为 v-model使用
    //变量名
    prop: "dateP",
    //事件名 自己定义事件名
    event: "date-choose",
  },
  methods: {
    //选择日期
    changeDate(date) {
      //输入框 赋值 调用主组件方法
      this.$emit("date-choose", date);
    },
  },
};

```

举例：v-model自定义事件和变量的使用，不用v-model实现
父组件

```

<!-- :dateP 为子组件定义的值 后面的dateP为父组件定义的值-->
<data-picker :dateP="dateP" @date-choose="changeValue"></data-picker>

```

```

import datePicker from "./DatePicker.vue";
export default {
  components: {
    DatePicker: datePicker,
  },
  data() {
    return {
      //给:dateP=dateP中后面的dateP赋值
      dateP: new Date(),
    };
  },
  methods: {
    //不用v-model时 datePicker会调用 给输入框赋值
    changeValue(date) {
      this.dateP = date;
    },
  },
};

```

子组件

```

export default {
  props: {
    //这个dateP就是父组件传来的值
    dateP: {
      type: Date,
      //默认值为数据或对象时 用函数返回
      //default: () => new Date(),
    },
  },
  methods: {
    //选择日期
    changeDate(date) {
      //输入框 赋值 调用主组件方法
      this.$emit("date-choose", date);
    },
  },
};

```

.sync 实现双向绑定

举例：

我们可以在后台使用vm.sText = 'xxx'给input框赋值，也可以通过vm.sText查询input框里的值

```

<sync-cmp :value.sync="sText"></sync-cmp>

//sync 一般用于input
Vue.component('sync-cmp', {
  props: ['value'],
  template: `
    <input :value='value' @input="$emit('update:value',$event.target.value)" />
  `
})
const vm = new Vue({
  el: '#app',
  data: {
    //sync
    sText: '',
  }
});

```

\$attrs 不推荐

祖先组件传递数据给子孙组件时，可以利用\$attrs传递。

demo或小型项目可以使用\$attrs进行数据传递，中大型项目不推荐，数据流会变的难于理解。

\$attrs的真正目的是撰写基础组件，将非Prop特性赋予某些DOM元素。

举例：\$attr 祖先组件传递数据给子孙组件

```
<attrs-cmp :inputvalue="fuvValue"></attrs-cmp>
```

```
//attrs 祖先组件传递数据给子孙组件
Vue.component('attrs-cmp', {
  //``inheritAttrs: false`` 和 ``$attrs`` 相互配合,可用于单选框
  props: ['label', 'value'],
  mounted() {
    console.log(this.$attrs);
  },
  template: `
    <label>
      {{ label }}
      <input
        v-bind:value="$attrs['inputvalue']"
        v-on:input="$emit('input', $event.target.value)"
      >
    </label>
  `,
})
const vm = new Vue({
  el: '#app',
  data: {
    //attrs
    fuvalue: '测试attrs',
  }
});
```

\$listeners

可以在子孙组件中执行祖先组件的函数，从而实现数据传递。

demo或小型项目可以使用\$listeners进行数据传递，中大型项目不推荐，数据流会变的难于理解。

\$listeners的真正目的是将所有的事件监听器指向这个组件的某个特定的子元素。

一般用于<label><input /></label>

举例：\$listeners的应用

```
<listeners-cmp @focus="onfocus" @blur="onblur"></listeners-cmp>

//listeners
Vue.component('listeners-cmp', {
  mounted() {
    console.log(this.$listeners);
  },
  template: `
    <label><input v-on="$listeners" /></label>
  `,
})
const vm = new Vue({
  el: '#app',
  methods: {
    //测试 $listeners
    onfocus() {
      console.log('focus');
    },
    onblur() {
      console.log('blur');
    }
  }
});
```

\$root

可以在子组件中访问根实例的数据。

对于 demo 或非常小型的有少量组件的应用来说这是很方便的。中大型项目不适用。会使应用难于调试和理解。

举例：\$root的使用

```

<root-cmp></root-cmp>

//$root 不推荐
Vue.component('root-cmp', {
  created() {
    console.log(this.$root.name);
  },
  template: `<div>测试root</div>`
})
const vm = new Vue({
  el: '#app',
  data: {
    //$/root
    name: '测试$root',
  },
  created() {
    console.log(this.$root.name);
  }
});

```

\$parent

可以在子组件中访问父实例的数据。

对于 demo 或非常小型的有少量组件的应用来说这是很方便的。中大型项目不适用。会使应用难于调试和理解。

举例：\$parent的使用

```

<parent-cmp></parent-cmp>

//$parent 不推荐
Vue.component('parent-cmp', {
  mounted() {
    //获取父组件中parentname的值
    console.log(this.$parent.parentname);
  },
  template: `<div>测试parent</div>`
})
const vm = new Vue({
  el: '#app',
  data: {
    //$/parent
    parentname: '测试$parent',
  }
});

```

\$children

可以在父组件中访问子实例的数据。

对于 demo 或非常小型的有少量组件的应用来说这是很方便的。中大型项目不适用。会使应用难于调试和理解。

举例：\$children的使用

```

<!-- $children 不推荐 -->
<children-cmp></children-cmp>

```

```

Vue.component('children-cmp', {
  data() {
    return {
      name: '我是子数据'
    }
  },
  template: `
    <div>测试children</div>
  `
})
const vm = new Vue({
  el: '#app',
  mounted() {
    console.log(this.$children[0].name);
  }
});

```

ref

可以在父组件中访问子实例的数据。

`refs`只会在组件渲染完成之后生效，并且它们不是响应式的，适用于`demo`或小型项目。举例：`refs`的应用

```
<ref-cmp ref="refCmp"></ref-cmp>
```

```

Vue.component('ref-cmp', {
  template: `
    <div>测试ref</div>
  `
})
const vm = new Vue({
  el: '#app',
  mounted() {
    //获取$refs.refCmp
    console.log(this.$refs.refCmp);
  }
});

```

provide & inject

祖先组件提供数据（`provide`），子孙组件按需注入（`inject`）。

`vm = new Vue(...)`里使用不了。

会将组件的阻止方式，耦合在一起，从而使组件重构困难，难以维护。不推荐在中大型项目中使用，适用于一些小组件的编写。

举例： `provide & inject`的使用

```
<zpi-cmp>
  <zpi-cmp></zpi-cmp>
</zpi-cmp>
</zpi-cmp>
```

```

//provide && inject
Vue.component('pi-cmp', {
  //需要加slot
  template: `
    <div>测试provide和inject<slot></slot></div>
  `,
  //提供数据
  provide() {
    return {
      share: '数据'
    }
  }
})
Vue.component('zpi-cmp', {
  template: `<div></div>`,
  //接收数据
  inject: ['share'],
  mounted() {
    console.log(this.share);
  }
})

```

eventBus(事件总线) 兄弟组件中传输数据 不推荐

举例：eventBus应用

```

//使用 eventBus(事件总线) 兄弟组件中传输数据
Vue.prototype.$bus = new Vue();
Vue.component('jj-cmp', {
  data() {
    return {
      a: '姐姐数据'
    }
  },
  template: `
    <button @click="onClick">测试eventBus(事件总线)</button>
  `,
  methods: {
    onClick() {
      //提供数据
      this.$bus.$emit('bus', this.a);
    }
  }
})
Vue.component('mm-cmp', {
  data() {
    return {
      mm: ''
    }
  },
  template: `<div>{{mm}}</div>`,
  mounted() {
    this.$bus.$on('bus', data => {
      //接收使用数据
      console.log(data);
      this.mm = data;
    })
  }
})

```

非父子组件通信时，可以使用这种方法，但仅针对于小型项目。中大型项目使用时，会造成代码混乱不易维护。

Vuex

状态管理，中大型项目时强烈推荐使用此种方式，日后再学~

混入

举例：混入的使用

```
<div id="app">
  <cmp-1></cmp-1>
  <cmp-2></cmp-2>
</div>

//全局混入
Vue.mixin({
  created() {
    console.log('我是全局组件，都要执行我');
    //this.$options 组件选项
    if (this.$options.xxx) {
      console.log('只有有xxx配件的才能执行我');
    }
  }
});
let mixin = {
  data() {
    return {
      //如果组件中有同名的用组件的
      a: '我是被混入了'
    }
  },
  mounted() {
    //如果组件中也有mounted方法，都会被执行
    console.log('我是被混入的方法');
  },
  methods: {
    fn() {
      //如果组件中有同名方法，用组件中方法
      console.log('我是被混入的method方法');
    }
  }
};
Vue.component('cmp2', {
  mixins: [mixin],
  xxx: '我因为全局混入被执行了',
  created() {
    console.log(this.a);
  },
  beforeMount() {
    this.fn();
  },
  mounted() {
    console.log('我是组件内方法');
  },
  data() {
    return {
      a: '我执行了，混入的就不执行了'
    }
  },
  methods: {
    fn() {
      console.log('我是组件内methods方法');
    }
  },
  template: `
    <div>cmp2</div>
  `
})
Vue.component('cmp1', {
  mixins: [mixin],
  created() {
    console.log(this.a);
  },
  template: `
    <div>cmp1</div>
  `
})
const vm = new Vue({
  el: '#app'
})
```

基础

混入 (mixin) 提供了一种非常灵活的方式，来分发 Vue 组件中的可复用功能。

一个混入对象可以包含任意组件选项。当组件使用混入对象时，所有混入对象的选项将被“混合”进入该组件本身的选项。

```
var mixin = {
  created () {
    this.hello();
  },
  methods: {
    hello () {
      console.log('hello, 我是混入中的函数');
    },
  }
}

Vue.component('my-cmp', {
  mixins: [mixin],
  template: `
    <div>xx</div>
  `
})
```

选项合并

当组件和混入对象含有同名选项时，这些选项会以恰当的方式进行“合并”。

合并数据，以组件数据优先：

```
var mixin = {
  data () {
    return {
      msg: 'hello',
    }
  }
}

new Vue({
  mixins: [mixin],
  data: {
    msg: 'goodbye',
  },
  created: function () {
    console.log(this.msg)
  }
})
```

合并钩子函数，将合并为一个数组。先调用混入对象的钩子，再调用组件自身钩子。

就是如果钩子函数相同，则会混入的和组件的钩子函数都只执行。

```
var mixin = {
  created () {
    console.log('混入对象钩子')
  }
}

new Vue({
  el: '#app',
  mixins: [mixin],
  created () {
    console.log('组件钩子')
  }
})
```

合并值为对象的选项，如 `methods`、`components` 等，将被合并为同一个对象。两个对象键名冲突时，取组件对象的键值对（就是比如`methods`中方法名一样时，会执行组件的方法）。

全局混入(谨慎使用)

混入也可以进行全局注册。使用时格外小心！一旦使用全局混入，它将影响每一个之后创建的 `Vue` 实例。使用恰当时，这可以用来为自定义选项注入处理逻辑。

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

谨慎使用全局混入，因为它会影响每个单独创建的 `Vue` 实例（包括第三方组件）。大多数情况下，只应当应用于自定义选项。

自定义指令

简介

我们可以自己写一个自定义指令去操作DOM元素，以达到代码复用的目的。注意，在 `Vue` 中，代码复用和抽象的主要形式是组件。然而，有的情况下，你仍然需要对普通 DOM 元素进行底层操作，这时候就会用到自定义指令。

全局注册指令：

```
Vue.directive('focus', {/** */})
```

局部注册指令

```
const vm = new Vue({  
  el: '#app',  
  directives: {  
    focus: {/** */}  
  }  
})
```

使用：

```
<input v-focus></input>
```

例如，写一个自动聚焦的输入框：

```
Vue.directive('focus', {  
  // 当被绑定的元素插入到DOM时执行  
  inserted: function (el) {  
    el.focus();  
  }  
})
```

此时，在`input`元素上使用 `v-focus` 指令就可以实现自动聚焦了。

钩子函数

自定义指令对象提供了钩子函数供我们使用，这些钩子函数都为可选。

bind

只调用一次，指令第一次绑定到元素时调用。在这里可以进行一次性的初始化设置。

inserted

被绑定元素插入父节点时调用(仅保证父节点存在, 但不一定已被插入文档中)。

update

所在组件的 VNode 更新时(页面更新时)调用, **但是可能发生在其子 VNode 更新之前**。

componentUpdated

所在组件的 VNode 更新时(页面更新时)调用, 指令所在组件的 VNode 及其子 VNode 全部更新后调用。

unbind

只调用一次, 指令与元素解绑时调用(被绑定的Dom元素被Vue移除)。

举例: 五种钩子方法使用

```
<div id="app">  
  <input type="text" v-focus:[num].number='test' v-if='is' />  
</div>  
  
//binding中arg(参数)为3 modifiers(修饰符)为.number  
Vue.directive('focus', {  
  //只调用一次, 指令第一次绑定到元素时调用。在这里可以进行一次性的初始化设置。  
  bind(el, binding) {  
    console.log('bind', el, binding);  
  },  
  //被绑定元素插入父节点时调用  
  inserted(el, binding) {  
    console.log('insert', el, binding);  
  },  
  //所在组件的 VNode 更新时(页面更新时)调用, 但是可能发生在其子 VNode 更新之前。  
  update(el, binding, vnode,oldVnode) {  
    console.log('update', el, binding, vnode,oldVnode);  
  },  
  //所在组件的 VNode 更新时(页面更新时)调用, 发生在其子 VNode 更新之后。  
  componentUpdated(el, binding, vnode,oldVnode) {  
    console.log('componentUpdated', el, binding, vnode,oldVnode);  
  },  
  //只调用一次, 指令与元素解绑时调用(被绑定的Dom元素被Vue移除)  
  //可用v-if尝试  
  unbind(el, binding, vnode) {  
    console.log('unbind', el, binding, vnode);  
  }  
})  
const vm = new Vue({  
  el: "#app",  
  data: {  
    test: '123',  
    //测试钩子方法unbind  
    is: true,  
    //slice参数  
    num:3  
  }  
)
```

钩子函数参数

- el: 指令所绑定的元素, 可以用来直接操作DOM。
- binding: 对象, 包含以下属性:
 - name: 指令名, 不包括 v- 前缀。
 - value: 指令的绑定值, 例如: v-my-directive="1 + 1" 中, 绑定值为 2。
 - oldValue: 指令绑定的前一个值, 仅在 update 和 componentUpdated 钩子中可用。无论值是否改变都可用。
 - expression: 字符串形式的指令表达式。例如 v-my-directive="1 + 1" 中, 表达式为 "1 + 1"。
 - arg: 传给指令的参数, 可选。例如 v-my-directive:foo 中, 参数为 "foo"。
 - modifiers: 一个包含修饰符的对象。例如: v-my-directive.foo.bar 中, 修饰符对象为 { foo: true, bar: true }。

- vnode: Vue 编译生成的虚拟节点。vnode.context 为 vm (vm= new Vue())
- oldVnode: 上一个虚拟节点, 仅在 update 和 componentUpdated 钩子中可用。

练习

模拟 v-show

```
// 绑定的值为false, display为none, 值为true, display为""
Vue.directive('myshow', {
  bind (el, binding, vnode, oldVnode) {
    var display = binding.value ? '' : 'none';
    el.style.display = display;
  },
  update (el, binding, vnode, oldVnode) {
    var display = binding.value ? '' : 'none';
    el.style.display = display;
  }
})
```

模拟 v-model

```
// 1. 通过绑定的数据, 给元素设置value
// 2. 当触发input事件时, 去更改数据的值
// 3. 更改数据后, 同步input的value值
Vue.directive('mymodel', {
  bind (el, binding) {
    const vm = vnode.context;
    const { value, expression } = binding;
    el.value = value;

    el.oninput = function (e) {
      const inputVal = el.value;
      vm[expression] = inputVal;
    }
  },
  update (el, binding) {
    const { value } = binding;
    el.value = value;
  }
})
```

写一个 v-slice (截取文本框)

```
Vue.directive('slice', {
  bind (el, binding, vnode) {
    const vm = vnode.context;
    let { value, expression, arg, modifiers } = binding;

    if(modifiers.number) {
      value = value.replace(/[^0-9]/g, '');
    }

    el.value = value.slice(0, arg);
    vm[expression] = value.slice(0, arg);

    el.oninput = function (e) {
      let inputVal = el.value;

      if(modifiers.number) {
        inputVal = inputVal.replace(/[^0-9]/g, '');
      }

      el.value = inputVal.slice(0, arg);
      vm[expression] = inputVal.slice(0, arg);
    }
  },
  update (el, binding, vnode) {
    const vm = vnode.context;
    let { value, arg, expression, modifiers } = binding;

    if(modifiers.number) {
      value = value.replace(/[^0-9]/g, '');
    }

    el.value = value.slice(0, arg);
    vm[expression] = value.slice(0, arg);
  },
})
```

动态指令参数

指令的参数可以是动态的。如: `v-directive:[arguments]="value , argument` 参数可以根据组件实例数据进行更新。

重写 v-slice

```

Vue.directive('slice', {
  bind (el, binding, vnode) {
    const vm = vnode.context;
    let { value, expression, arg, modifiers } = binding;

    if(modifiers.number) {
      value = value.replace(/[^0-9]/g, '');
    }

    el.value = value.slice(0, arg);
    vm[expression] = value.slice(0, arg);

    el.oninput = function (e) {
      let inputVal = el.value;

      if(modifiers.number) {
        inputVal = inputVal.replace(/[^0-9]/g, '');
      }

      el.value = inputVal.slice(0, arg);
      vm[expression] = inputVal.slice(0, arg);
    }
  },
  update (el, binding, vnode) {
    const vm = vnode.context;
    let { value, arg, expression, modifiers } = binding;

    if(modifiers.number) {
      value = value.replace(/[^0-9]/g, '');
    }

    el.value = value.slice(0, arg);
    vm[expression] = value.slice(0, arg);

    el.oninput = function (e) {
      let inputVal = el.value;

      if(modifiers.number) {
        inputVal = inputVal.replace(/[^0-9]/g, '');
      }

      el.value = inputVal.slice(0, arg);
      vm[expression] = inputVal.slice(0, arg);
    }
  },
})

```

函数简写

当想在 `bind` 和 `update` 中触发相同行为，而不关心其他钩子时，可以写成函数的形式：

```

Vue.directive('myshow', (el, binding) => {
  const { value } = binding;
  const display = value ? '' : 'none';
  el.style.display = display;
})

```

```

Vue.directive('slice', (el, binding, vnode) => {
  const vm = vnode.context;
  let { value, expression, arg, modifiers } = binding;

  if(modifiers.number) {
    value = value.replace(/[^0-9]/g, '');
  }

  el.value = value.slice(0, arg);
  vm[expression] = value.slice(0, arg);

  el.oninput = function (e) {
    let inputVal = el.value;

    if(modifiers.number) {
      inputVal = inputVal.replace(/[^0-9]/g, '');
    }

    el.value = inputVal.slice(0, arg);
    vm[expression] = inputVal.slice(0, arg);
  }
})

```

对象字面量

如果自定义指令需要多个值，可以传入一个 JS 对象字面量。指令函数能够接受所有合法的 JS 表达式。

```

<div v-demo="{ color: 'white', text: 'hello!' }"></div>

Vue.directive('demo', function (el, binding) {
  console.log(binding.value.color) // => "white"
  console.log(binding.value.text) // => "hello!"
})

```

过滤器

自定义过滤器，用于一些常见的文本格式化。

过滤器可用在两个地方：双花括号插值 和 v-bind 表达式，添加在JS表达式的尾部，由“管道”符号表示：

```

<!-- 在双花括号中 --&gt;
{{ message | filter }}

<!-- 在 v-bind 中 --&gt;
&lt;div v-bind:id="id | filter"&gt;&lt;/div&gt;
</pre>

```

定义过滤器

全局过滤器：

```
Vue.filter('filter', value => {})
```

局部过滤器：

```
<div id="app">{{ content | myfilter(0.5) }}</div>
```

```

const vm = new Vue({
  el: "#app",
  data: {
    content: '我是测试过滤器的'
  },
  filters: {
    //过滤器
    myfilter(val, param) {
      console.log(val, param);
      return '我是内部的，比全局更先执行'
    }
  }
})

```

参数

当过滤器形式为 `msg | filter` 时, filter过滤器接收一个参数, 参数为 `msg`。

当过滤器形式为 `msg | filter('a')` 时, filter过滤器接收两个参数, 参数为 `msg, 'a'`

过滤器串联

```
<div id="app">{{ content | myfilter(0.5) | myfilter1 }}</div>
```

在这个例子中, `myfilter`的参数为 `content` , `myfilter1`的参数为`myfilter`。

```

const vm = new Vue({
  el: "#app",
  data: {
    content: '我是测试过滤器的'
  },
  filters: {
    //过滤器 串联 myfilter myfilter1
    myfilter(val, param) {
      console.log(val, param);
      return '我是内部的，比全局更先执行'
    },
    myfilter1(val) {
      console.log(val);
      return '过滤器串联，我只看我前面的值'
    }
  }
})

```

练习

首字母大写

```
{{ content | capitalize }}
```

```
Vue.filter('capitalize', value => {
  if(!value) { return }; 
  return value.charAt(0).toUpperCase() + value.slice(1);
})
```

数字中间加上逗号

```
{{ money | toMoney }}
```

```
Vue.filter('toMoney', value => {
  if(!value) { return };
  return value.toLocaleString();
});
```

数字添加文字“万”

```
{} likes | addWord }

Vue.filter('addWord', value => {
  if(!value) { return };
  if(value > 10000) {
    return ( value / 10000).toFixed(1) + '万';
  }
  return value;
});
```

安装脚手架

安装@vue/cli

node 版本要求: >8.9, 推荐使用8.11.0 +。

关于旧版本:

如果在这之前已经全局安装了旧版本的vue-cli(1.x 或 2.x), 那么需要先卸载掉。

运行: `npm uninstall vue-cli -g` 或 `yarn global remove vue-cli`。

安装:

```
npm install -g @vue/cli
# OR
yarn global add @vue/cli
```

安装之后, 可以在命令行中访问vue命令。

检查版本是否正确:

```
vue --version
```

快速原型开发

安装:

```
npm install -g @vue/cli-service-global
# OR
yarn global add @vue/cli-service-global
```

组件结构

启动组件: 进入App.vue父级文件夹终端, 输入`vue serve`, 这时, 就会启动, 终端会给出访问地址, 这个地址就是我们访问App.vue的地址, 其他vue模块可以往App.vue里面加, 如果想访问其他.vue文件那么在终端舒服`vue serve xxx.vue`

举例: App.vue

```
<template>
<div>
  {{ content }}
  <base-son />
</div>
</template>
<script>
import son from "./Son";
export default {
  components: {
    baseSon: son,
  },
  data() {
    return {
      content: "我是妹妹",
    };
  },
};
</script>
<style>
div {
  background-color: azure;
}
</style>
```

我们可能会在son.vue中定义样式，如果想样式仅在Son.vue中使用，需要在style中加入scoped `<style scoped>` ,如果不加，引用Son.vue相当于在全局加了Son.vue的样式

Son.vue

```
<template>
<div>
  son
  <p>hahaah is a p</p>
</div>
</template>

<style scoped>
p {
  background-color: green;
}
</style>
```

组件中

用 `<template>...</template>` 放入html

用 `<script>...</script>` 放入js

用 `<style></style>` 放入样式

安装vscode插件

名字：Vetur。用于高亮.vue文件代码

练习_树形组件

数据：

```

data: [
  {
    label: "一级 1",
    children: [
      {
        label: "二级 1-1",
        children: [
          {
            label: "三级 1-1-1"
          }
        ]
      }
    ]
  },
  {
    label: "一级 2",
    children: [
      {
        label: "二级 2-1",
        children: [
          {
            label: "三级 2-1-1"
          }
        ]
      },
      {
        label: "二级 2-2",
        children: [
          {
            label: "三级 2-2-1"
          }
        ]
      }
    ]
  },
  {
    label: "一级 3",
    children: [
      {
        label: "二级 3-1",
        children: [
          {
            label: "三级 3-1-1"
          }
        ]
      },
      {
        label: "二级 3-2",
        children: [
          {
            label: "三级 3-2-1"
          }
        ]
      }
    ]
  }
]

```

利用脚手架搭建项目

有两种方式搭建项目(我们用的是运行时版, 体积比完整版少了30%)

(不推荐) 第一种终端进入要搭建项目文件夹, 输入vue ui进入图形化项目搭建页面, 搭建项目。

(推荐) 第二种终端进入要搭建项目文件夹, 在终端内直接建立项目。

第二种方法步骤:

- ①在终端输入 vue create xxx(项目名字) 注意项目名字中不能有大写字母
- ②选择预设选项 (用空格选择, 选好按回车) (其中有一个选项 Linter/Formatter为要求团队中代码风格都一致的选项)
- ③选择配置要放在哪里, 在这里, 会选择, 为未来的项目是否保存预设, 这个主要是看未来项目预设是否同当前项目相同, 如果相同就保存, 不相同就不保存, 如果保存, 会提示我们, 需要给预设起个名字
- ④项目会启动
- ⑤会提示我们 cd xxx (进入项目文件夹) 然后 npm run serve (开启服务)

这时我们的项目已经生成，项目内包含的文件夹解析：

```
node_modules :项目依赖包文件  
public: 默认的打开的html文件就在这，在这个html文件中我们会注入js，js文件(main.js)在src文件夹中。  
src:  
1.main.js主入口文件，其中有句render: h => h(App)是将 vue嵌入到对应的html文件中。  
2.App.vue组件中入口  
3.assets文件夹放置css文件图片文件等静态文件  
4.components文件放置组件  
.gitignore:记录有哪些文件不需要上传github  
package.json
```

如果想添加webpack配置，则自己新建vue.config.js，在这个文件中添加想添加的内容。

注意在这个两种方式中，我们自己定义的预设会存进电脑>用户>.vuerc文件中如果想删除预设，就把文件中预设部分删除就可以。

拉取 2.x 模板（旧版本）

```
npm install -g @vue/cli-init  
# `vue init` 的运行效果将会跟 `vue-cli@2.x` 相同  
vue init webpack my-project
```

渲染函数

基础

当我们需要使用JavaScript的编程能力时，可以利用渲染函数。渲染函数比模板更接近于编译器。

例如，我们想要生成一些标题：

```
<h1>Hello world!</h1>
```

如果，我们按照之前的方式，那么模板内将会十分冗余。如果此时利用渲染函数，那么代码写起来将会简洁很多。

```
props: {  
  level: {  
    type: Number,  
    required: true  
  }  
},  
render: function (createElement) {  
  return createElement(  
    'h' + this.level, // 标签名称  
    this.$slots.default // 子节点数组  
  )  
},
```

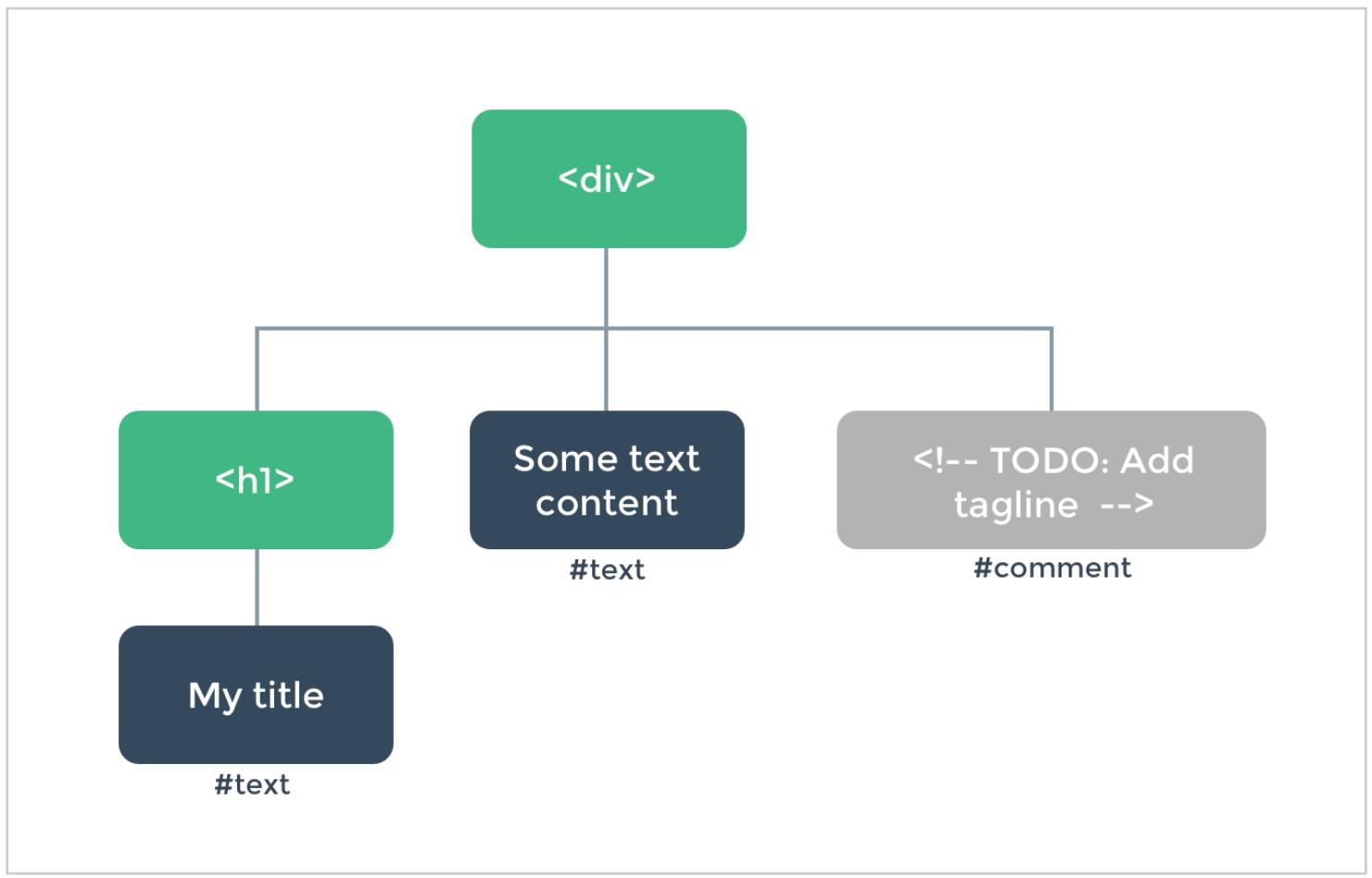
render函数的优先级：在new Vue中的template选项 < render < 在.vue实例中的写结构用的 <template></template>

节点、树、以及虚拟DOM

在深入渲染函数之前，先来了解一些浏览器的工作原理。例如，下面这段HTML：

```
<div>  
  <h1>My title</h1>  
  Some text content  
  <!-- TODO: Add tagline -->  
</div>
```

当浏览器读到这些代码时，它会建立一个**DOM节点树** 来保持追踪所有内容，如同你会画一张家谱树来追踪家庭成员的发展一样。
以上HTML对应的DOM节点树如下图所示：



每个元素都是一个节点。每段文字也是一个节点。甚至注释也都是节点。一个节点就是页面的一个部分。就像家谱树一样，每个节点都可以有孩子节点。

高效地更新所有这些节点是比较困难的，不过幸运的是，我们不需要手动完成这个工作。只需要告诉Vue希望页面上的HTML是什么，例如在模板中：

```
<h1>{{ blogTitle }}</h1>
```

或者是在一个渲染函数中：

```
render: function (createElement) {
  return createElement('h1', this.blogTitle)
}
```

在这两种情况下，Vue 都会自动保持页面的更新，即便 `blogTitle` 发生了改变。

虚拟DOM

Vue通过建立一个虚拟DOM来追踪自己要如何改变真实DOM。例如：

```
return createElement('h1', this.blogTitle);
```

`createElement` 会返回什么呢？

它不会返回一个实际的DOM元素。更准确的名字可能是： `createNodeDescription`，因为它所包含的信息会告诉Vue页面上需要渲染什么样的节点，包括其子节点的描述信息。我们把这样的节点描述为“虚拟节点（virtual node）”，也常简写它为“VNode”。“虚拟DOM”是我们对由Vue组件树建立起来的整个VNode树的称呼。

createElement参数

`createElement`接收的参数：

```
createElement(标签名(必需), 与模板中属性对应的数据对象(可选), 子级虚拟节点(可选));
```

标签名: html标签或组件

与模板中属性对应的数据对象: 描述对象使用, 比如class是什么, id是什么, 还可以是html标签数组, 或是字符串, 是字符串时就是内部的内容就是个文本节点

子级虚拟节点: 字符串或数组, 如果是字符串, 就证明子级是个文本节点。

深入 ‘与模板中属性对应的数据对象’

```
{
  // 与 `v-bind:class` 的 API 相同，接受一个字符串、对象或字符串和对象组成的数组
  class: {
    foo: true,
    bar: false
  },
  // 与 `v-bind:style` 的 API 相同，接受一个字符串、对象，或对象组成的数组
  style: {
    color: 'red',
    fontSize: '14px',
  },
  // 普通的 HTML attribute
  attrs: {
    id: 'foo',//给组件赋id='foo'
  },
  // 组件 prop (父级传来的变量)
  props: {
    myProp: 'bar',
  },
  //特性和属性：特性是写在行间的，属性是通过dom元素点出来的那些值
  // DOM属性
  domProps: {
    innerHTML: 'baz',
  },
  // 事件监听器，不支持如“v-on:keyup.enter”这样的修饰器
  on: {
    click: this.onClick
  },
  // 仅用于组件，用于监听原生事件，而不是组件内部使用 vm.$emit 触发的事件。
  nativeOn: {
    click: this.nativeClickHandler
  },
  // 使用自定义指令。注意，无法对 `binding` 中的 `oldValue` 赋值，因为 Vue 已经自动为你进行了同步。
  directives: [
    {
      name: 'my-custom-directive',
      value: '2',//或是this.xxx
      expression: '1 + 1',//表达式的值
      arg: 'foo',//自定义指令的参数 比如 v-slice:5 中的5
      modifiers: {
        bar: true
      }
    }
  ],
  //使用自定义指令举例 <input v-slice:5.number="content" />
  // directives: [
  //   {
  //     name: 'slice', // v-slice
  //     value: this.content, // v-slice="content"
  //     expression: 'content',
  //     arg: 5, // v-slice:5="content"
  //     modifiers: {
  //       number: true // v-slice:5.number="content"
  //     },
  //   }
  // ]
  // 其它特殊顶层属性
  key: 'myKey',
  ref: 'myRef',
  // 如果在渲染函数中给多个元素都应用了相同的 ref 名，那么 `$refs.myRef` 会变成一个数组。(就是说想让$refs.myRef的值变为数组，refInFor的值就得为true)
  refInFor: true
  // 作用域插槽，格式为: { name: props => VNode | Array<VNode> }
  // 如果组件是其它组件的子组件，需为插槽指定名称
  slot: 'name-of-slot',
  scopedSlots: {
    default: props => createElement('span', props.text)
  },
}
}
```

使用JavaScript代替模板功能

v-if 和 v-for

只要在原生的 JavaScript 中可以轻松完成的操作，Vue 的渲染函数就不会提供专有的替代方法。比如，在模板中使用的 v-if 和 v-for：

```
<ul v-if="items.length">
  <li v-for="item in items">{{ item }}</li>
</ul>
<p v-else>No items found.</p>
```

这些都可以在渲染函数中用 JavaScript 的 if/else 和 map 来重写：

```
props: ['items'],
render (createElement) {
  if(items.length) {
    return createElement('ul', this.items.map(item => createElement('li', item)))
  } else {
    return createElement('p', 'No items found');
  }
}
```

v-model

渲染函数中没有与v-model的直接对应---必须自己实现相应的逻辑：

```
data () {
  return {
    value: 'ceshi',
  }
},
render (createElement) {
  const self = this;
  return createElement('input', {
    attrs: {
      value: self.value
    },
    on: {
      input (e) {
        self.value = e.target.value;
      }
    },
  });
},
```

事件&按键修饰符

对于 .passive、.capture 和 .once 这些事件修饰符，Vue 提供了相应的前缀可以用于 on：

修饰符	前缀
.passive	&
.capture	!
.once	~
.capture.once 或 .once.capture	~!

例如：

```
on: {
  '!click': this.doThisInCapturingMode,
  '~keyup': this.doThisOnce,
  '~!mouseover': this.doThisOnceInCapturingMode
}
```

对于所有其它的修饰符，私有前缀都不是必须的，因为你在事件处理函数中使用事件方法：

修饰符	处理函数中的等价操作
.stop	event.stopPropagation()

修饰符	处理函数中的等价操作
.prevent	event.preventDefault()
.self	if (event.target !== event.currentTarget) return
按键: .enter, .13	if (event.keyCode !== 13) return 对于别的按键修饰符来说, 可将 13 改为另一个按键码
修饰键: .ctrl, .alt, .shift, .meta	if (!event.ctrlKey) return (将 ctrlKey 分别修改为 altKey、shiftKey 或者 metaKey)

插槽

可以通过 `this.$slots` 访问静态插槽的内容, 每个插槽都是一个 `VNode` 数组:

```
<div>
  <slot></slot>
</div>

render: function (createElement) {
  return createElement('div', this.$slots.default)
}
```

也可以通过 `this.$scopedSlots` 访问作用域插槽, 每个作用域插槽都是一个返回若干 `VNode` 的函数:

```
<div>
  <slot :text="message"></slot>
</div>

data() {
  return {
    msg: 'hello world',
  }
},
render: function (createElement) {
  return createElement('div', [
    this.$scopedSlots.default({
      text: this.msg
    })
  ])
}
```

如果要用渲染函数向子组件中传递作用域插槽, 可以利用 `VNode` 数据对象中的 `scopedSlots` 字段:

```
<div>
  <base-slot v-slot="slotProps">
    {{ slotProps.text }}
  </base-slot>
</div>

render: function (createElement) {
  return createElement('div', [
    createElement('base-slot', {
      // 在数据对象中传递 `scopedSlots`
      // 格式为 { name: props => VNode | Array<VNode> }
      scopedSlots: {
        default: function (props) {
          return createElement('span', props.text)
        }
      }
    })
  ])
}
```

JSX (JS + XML(html) == JSX)

为了让render函数更好理解，所以在Vue中使用JSX语法。可以让我们回到更接近模板的语法上。

```
render () {
  return (
    <h1>这是一个标题</h1>
  )
}
```

插值

在JSX中我们一般用<>去括html，用{}去括js

```
<div>{ this.value }</div>
```

举例：

```
export default {
  name: "App",
  data() {
    return { value: "hello world" };
  },
  components: {
    HelloWorld,
  },
  render() {
    return <h1>{this.value}</h1>;
  },
};
```

指令

在JSX中，一些指令并不存在，所以我们可以换一种方式来处理。

v-text

就是在domProps前缀加上想改变的内容，比如domPropsTextContent 是改变文案内容的，但是这个在有些版本不好使

```
export default {
  name: "App",
  render() {
    return (
      <h1>
        <div domPropsTextContent="<p>i am a p</p>"></div>
      </h1>
    );
  },
};
```

结果：打印出 <p>i am a p</p>

v-html

就是在domProps前缀加上想改变的内容，比如domPropsInnerHTML 是改变innerHTML的，但是这个在有些版本不好使

```
export default {
  name: "App",
  data() {
    return { value: "hello world" };
  },
  render() {
    return (
      <h1>
        <div domPropsInnerHTML=<a>href</a>"></div>
      </h1>
    );
  },
};
```

v-show(控制是否显示)

jsx支持v-show指令：显示：true 不显示false

```
export default {
  name: "App",
  data() {
    return { show: false };
  },
  render() {
    return (
      <h1>
        <div v-show={this.show}>我是v-show</div>
      </h1>
    );
  },
};
```

v-if

jsx中不支持v-if, v-if....else....，所以我们要想使用需要自己模拟，但是v-if...else if...else...这种类型没法用简单的语句来模拟，需要通过函数

```
<!-- 模拟v-if 显示或不显示有前面的true/false决定 -->
{true && <div>div</div>}
<!-- 模拟v-if ...else .... 显示或不显示有前面的true/false决定 -->
{true ? <div>div</div> : <span>span</span>}
```

```

export default {
  name: "App",
  data() {
    return {
      //控制if
      num: 2,
    };
  },
  methods: {
    //模拟v-if ..else if... else...
    vif() {
      if (this.num === 1) {
        return "<div>if</div>";
      } else if (this.num === 2) {
        return "<div> else if</div>";
      } else {
        return "<div> else</div>";
      }
    },
  },
  render() {
    return (
      <h1>
        v-if {false && <div>v-if</div>}
        v-if.. else ..{true ? <div>v-if</div> : <div>else</div>}
        v-if.. else if ..else..{this.vif()}
      </h1>
    );
  },
};

```

v-for

jsx不支持，需要模拟，且每个for元素里需要加key，提升页面性能

```
{ [1, 2, 3].map(item => (<div key={item}>{ item }</div>))}
```

举例：应用

```

export default {
  name: "App",
  render() {
    return (
      <h1>
        {[1, 2, 3].map((item) => (
          <div key={item}>{item}</div>
        )))
      </h1>
    );
  },
};

```

v-on

就是on加上事件，要小驼峰式，比如onClick。或on-事件名（这都要求是小写），组件原生事件是nativeOn加上事件名，要小驼峰式，比如nativeOnClick

```

<button onClick={this.handleClick}>点击事件</button>
<button on-click={this.handleClick}>点击事件</button>
<!-- 对应@click.native 组件的原生事件--&gt;
&lt;cmp-button nativeOnClick={this.handleClick}&gt;&lt;/cmp-button&gt;
<!-- 传递参数 --&gt;
&lt;button onClick={e =&gt; this.handleClick(this.id)}&gt;触发点击事件时，传递参数&lt;/button&gt;
</pre>

```

举例：

```

import CmpButton from "./components/CmpButton";
export default {
  name: "App",
  components: {
    CmpButton,
  },
  methods: {
    handlerClick(num) {
      console.log(num);
    },
  },
  render() {
    return (
      <h1>
        <button onClick={this.handlerClick}>onClick</button>
        <button on-click={this.handlerClick}>on-click</button>
        <button onClick={(e) => this.handlerClick(2)}>click传参</button>
        <cmp-button nativeOnClick={this.handlerClick}></cmp-button>
      </h1>
    );
  },
};

```

CmpButton组件

```

<template>
  <div>组件</div>
</template>

```

v-bind

```
<input value={this.value} />
```

应用：

```

export default {
  name: "App",
  data() {
    return {
      //控制显示内容
      content: "v-bind",
    };
  },
  render() {
    return (
      <h1>
        <input value={this.content} />
        <div class={"a", "b"} style={{ fontSize: "14px", color: "red" }}>
          v-bind
        </div>
      </h1>
    );
  },
};

```

在JSX中可以直接使用class="xx"来指定样式类,内联样式可以直接写成style="xxx",以下是格式要求

```

<div class="a b" style="font-size: 12px;">Content</div>
<div class={"a", "b"} style="font-size: 12px;">Content</div>
<div class={{a: true, b: false}}>Content</div>
<div style={{color: 'red', fontSize: '14px'}}>Content</div>

```

v-model

有相应的插件 支持 v-model, 所以可以直接使用：

```
<input type="text" v-model={this.value} />
```

应用：

```

export default {
  name: "App",
  data() {
    return {
      //控制显示内容
      content: "v-bind",
    };
  },
  render() {
    return (
      <h1>
        <input type="text" v-model={this.content} />
      </h1>
    );
  },
};

```

v-pre

v-cloak

v-once

以上三个指令，不常用，无替代方案

Ref

从`this.$refs.xxx` 中获取ref的值

```
<div ref="xxx">xxx</div>
```

应用：

```

export default {
  name: "App",
  render() {
    return (
      <h1>
        <div ref="ourref">ref</div>
        {[4, 5, 6].map((item) => (
          <div ref="xx" refInFor={true} key={item}>
            {item}
          </div>
        )));
      </h1>
    );
  },
  mounted() {
    console.log(this.$refs.ourref);
    console.log(this.$refs.xx);
  },
};

```

当遍历元素或组件时，如：

```

{[4, 5, 6].map((item) => (
  <div ref="xx" key={item}>
    {item}
  </div>
));
}

```

会发现从`this.$refs.xxx` 中获取的并不是期望的数组值，此时就需要将`refInFor`属性设置为`true`了：

```

[[4, 5, 6].map((item) => (
  <div ref="xx" refInFor={true} key={item}>
    {item}
  </div>
))}
```

自定义指令（没有尝试，对错未定）

```

render () {
  // 1
  return (
    <input v-splice={{value: this.value, modifiers: {number: true }}}/>
  )

  // 2
  const directives = [
    {
      name: 'splice',
      value: this.value,
      modifiers: {number: true }
    }
  ];
}

return (
  <div {...{ directives}}></div>
)
}
```

过滤器

```

<!-- 正常使用过滤器 -->
<div>{{ value | myfilter }}</div>

<!-- 在jsx中使用过滤器 -->
<div>{this.$options.filters["myfilter"](this.value)}</div>
```

应用举例：

```

export default {
  name: "App",
  data() {
    return {
      value: "111"
    };
  },
  filters: {
    //过滤器
    myfilter(val) {
      console.log(val);
      return "我是内部的，比全局更先执行";
    },
  },
  render() {
    return (
      <h1>
        <div>{this.$options.filters["myfilter"]({this.value})}</div>
      </h1>
    );
  }
};
```

插槽

模板写法：

```

<!-- 子组件内 -->
<template>
  <div><slot name="footer"></slot></div>
</template>

<!-- 父组件 使用时 -->
<template>
  <div>
    <b-cop>
      <template v-slot:header>头部</template>
      <template v-slot:footer>
        <p>底部</p>
      </template>
    </b-cop>
  </div>
</template>

```

JSX写法:

```

<!-- 子组件内 -->
export default {
  render() {
    return (
      <div>
        {this.$slots.default}
        <p>{this.$slots.header}</p>
      </div>
    );
  },
};

<!-- 父组件 使用时 -->
import CmpSlot from "./components/CmpSlot.vue";
export default {
  name: "App",
  components: {
    CmpSlot,
  },
  render() {
    return (
      <h1>
        <cmp-slot>
          default
          <template slot="header">header</template>
        </cmp-slot>
      </h1>
    );
  }
};

```

作用域插槽:

模板写法:

```

<!-- 子组件内 -->
<template>
  <div>
    1111
    <slot :text="'HelloWorld'"></slot>
  </div>
</template>

```

```

<!-- 父组件 使用时 -->
<template>
  <div>
    <b-com v-slot="slotProps">
      {{ slotProps.text }}
    </b-com>
  </div>
</template>

```

JSX写法:

```
<!-- 子组件内 -->
<div class="demo">
  {
    this.$scopedSlots.default({
      text: 'HelloWorld',
    })
  }
</div>
```

```
<!--父级组件 使用时 -->
<div id="app">
  <base-demo {...{
    scopedSlots: {
      default: props => props.text
    },
  }}></base-demo>
</div>
```

使用: 父组件

```
//第一种写法
import CmpSlot from "./components/CmpSlot.vue";
export default {
  name: "App",
  components: {
    CmpSlot,
  },
  render() {
    return (
      <h1>
        <cmp-slot
          {...{
            scopedSlots: {
              default: (props) => props.text,
            },
          }}
        ></cmp-slot>
      </h1>
    );
  },
};
```

```
//第二种写法
import CmpSlot from "./components/CmpSlot.vue";
export default {
  name: "App",
  components: {
    CmpSlot,
  },
  render() {
    const scopedSlots = {
      scopedSlots: {
        default: (props) => <span>{props.text}</span>,
      },
    };
    return (
      <h1>
        <cmp-slot {...scopedSlots}></cmp-slot>
      </h1>
    );
  },
};
```

子组件:

```

export default {
  render() {
    return (
      <div>
        {this.$scopedSlots.default({
          text: "~~~HelloWorld~~~~~",
        })}
      </div>
    );
  },
};

```

函数式组件

当一个组件不需要状态（即响应式数据）、不需要任何生命周期场景、只接受一些props来显示组件时，我们可以将其标记为函数式组件。

```
functional: true,
```

举例（子组件）：

```

export default {
  props: ["level"],
  functional: true,
  render(h, context) {
    //context 为上下文，提供数据
    const props = context.props;
    const tag = "h" + props.level;
    return <tag></tag>;
  },
};

```

因为函数式组件只是函数，所以渲染开销会低很多。

在 2.3.0 之前的版本中，如果一个函数式组件想要接收 prop，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 attribute 都会被自动隐式解析为 prop。

为了弥补缺少的实例，render 函数提供第二个参数context作为上下文。context包括如下字段：

- props: 提供所有 prop 的对象
- slots: 一个函数，返回了包含所有插槽(非作用域)的对象
- scopedSlots: (2.6.0+) 一个暴露传入的作用域插槽的对象。也以函数形式暴露普通插槽。
- data: 传递给组件的整个数据对象，作为 createElement 的第二个参数传入组件，其中对象data.attrs中会显示prop中的数据，但是如果是props显示声明的数据就不会显示在data.attrs中，只有props隐式声明的才会显示在attrs中。
- parent: 对父组件的引用
- listeners: (2.3.0+) 一个包含了所有父组件为当前组件注册的事件监听器的对象。这是 data.on 的一个别名。
- injections: (2.3.0+) 如果使用了 inject 选项，则该对象包含了应当被注入的属性。

举例应用：

```

//子组件
export default {
  // props: ["level"],
  functional: true,
  //需要再次先数据，context.injections才会有值
  inject: ["name"],
  render(h, context) {
    //提供所有 prop 的对象
    const props = context.props;
    //一个函数，返回了包含所有插槽(非作用域)的对象
    const slot = context.slots().default;
    //一个暴露传入的作用域插槽的对象。也以函数形式暴露普通插槽。
    const header = context.scopedSlots.header()[0];
    //对象data.attrs中会显示prop中的数据,
    //但是如果props显示声明的数据就不会显示在data.attrs中,
    //只有props隐式声明的才会显示在attrs中
    const levelObj = context.data.attrs;
    //父级 组件
    const parent = context.parent;
    //一个包含了所有父组件为当前组件注册的事件监听器的对象，可把listeners
    //换为data.on也可以
    const listener = context.listeners;
    const listener1 = context.data.on;
    //父级的注入值 父级注入了，自己在收一下，context.injections才有值
    const injections = context.injections;

    const tag = "h" + props.level;
    return <tag></tag>;
  },
};

//父组件
<template>
  <level-cmp :level="2" @click="count++">
    ok
    <template v-slot:header>头部</template>
  </level-cmp>
</template>
<script>
  import LevelCmp from "./components/LevelCmp";
  export default {
    name: "Bpp",
    //要先注入，子组件inject才能获取到值
    provide() {
      return {
        name: "我是注入的",
      };
    },
    components: {
      LevelCmp,
    },
    data() {
      return {
        count: 0,
        share: "我是注入的",
      };
    },
  };
</script>

```

- children: VNode 子节点的数组，包含了所有的非作用域插槽和非具名插槽。具名为default <template v-slot:default>头部</template> 的都不会显示

slots() VS children

示例1：

```
<base-level :level="1" @click="handleClick">

<template v-slot:header>
  <div>我是头部</div>
</template>

<div>div</div>
<p>p</p>
<template>template</template>

</base-level>
```

slots() 的结果为：

```
{
  default:[<div>div</div>, <p>p</p>, template],
  header: [<div>我是头部</div>]
}
```

children 的结果为：

```
[<div>div</div>, <p>p</p>, template]
```

示例2：

```
<base-level :level="1" @click="handleClick">

<template v-slot:header>
  <div>我是头部</div>
</template>

<template v-slot:default>
  <div>div</div>
</template>

<p>p</p>
<template>template</template>

</base-level>
```

slots() 的结果为：

```
{
  default:[<div>div</div>],
  header: [<div>我是头部</div>]
}
```

children 的结果为：

```
[<div>div</div>, <p>p</p>, template]
```

基于模板的函数式组件

在 2.5.0 及以上版本中，如果你使用了单文件组件，那么基于模板的函数式组件可以这样声明：

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

过渡_单元素过渡

Vue 在插入、更新或者移除 DOM 时，提供多种不同方式的应用过渡效果。

单元素/组件的过渡

Vue 提供了 `transition` 的封装组件，在下列情形中，可以给任何元素和组件添加进入/离开过渡

- 条件渲染（使用 `v-if`）
- 条件展示（使用 `v-show`）
- 动态组件
- 组件根节点

过渡的类名

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`:

定义进入过渡的结束状态(2.1.8+)。

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。

在元素被删除之前生效 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`:

定义离开过渡的结束状态(2.1.8+)。

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。

举例（组件）： `` `html

图示：

```
<p class="markdown-p-center">
  
</p>
<p class="markdown-img-description">
  过渡
</p>
```

```
## 类名前缀
<p class="mume-header " id="类名前缀"></p>
```

1. transition 无 name 特性

类名前缀为 v-。

2. transition 有 name 特性

如 name 为 fade，则类名前缀为 fade-。

举例（组件）：

```
```html
<template>
 <div>
 <button @click="show = !show">click</button>
 <transition name="box1">
 <div class="box" v-show="show">i am ok</div>
 </transition>
 <transition name="box2">
 <div class="box" v-show="show">i am ok</div>
 </transition>
 </div>
</template>
<script>
export default {
 data() {
 return {
 show: true,
 };
 },
}
</script>
<style scoped>
.box {
 margin-top: 30px;
 width: 150px;
 height: 150px;
 border: 1px solid red;
 text-align: center;
 line-height: 150px;
}
.box1-leave,
.box1-enter-to {
 opacity: 1;
}
.box1-leave-active,
.box1-enter-active {
 transition: opacity 3s;
}
.box1-leave-to,
.box1-enter {
 opacity: 0;
}

.box2-leave,
.box2-enter-to {
 opacity: 1;
}
.box2-leave-active,
.box2-enter-active {
 transition: opacity 3s;
}
.box2-leave-to,
.box2-enter {
 opacity: 0;
}
```

```
}
```

```
</style>
```

## CSS 动画

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。

举例（组件）：应用

```
<template>
 <div>
 <button @click="show = !show">click</button>
 <transition>
 <div class="box" v-show="show">i am ok</div>
 </transition>
 </div>
</template>
<script>
export default {
 data() {
 return {
 show: true,
 };
 },
</script>
<style scoped>
.box {
 margin-top: 30px;
 width: 150px;
 height: 150px;
 border: 1px solid red;
 text-align: center;
 line-height: 150px;
}
/* .v-leave,
.v-enter-to {
 opacity: 1;
} */
.v-leave-active {
 animation: animate 1s reverse;
}

.v-enter-active {
 animation: animate 1s;
}

/* .v-leave-to,
.v-enter {
 opacity: 0;
} */
@keyframes animate {
 0% {
 opacity: 0;
 transform: translateX(400px) scale(1);
 }
 50% {
 opacity: 0.5;
 transform: translateX(200px) scale(1.5);
 }
 100% {
 opacity: 1;
 transform: translateX(0px) scale(1);
 }
}
</style>
```

## 自定义过渡的类名

我们可以通过以下 `attribute` 来自定义过渡类名：

- enter-class
- enter-active-class
- enter-to-class (2.1.8+)
- leave-class
- leave-active-class
- leave-to-class (2.1.8+)

他们的优先级高于普通的类名，这对于 Vue 的过渡系统和其他第三方 CSS 动画库（如 Animate.css）结合使用十分有用。Animate.css的版本必须为 ^3.7.2

Animate.css 官网地址: <https://daneden.github.io/animate.css/>

安装方式: `npm install animate.css --save`

举例（组件）：应用

```
<template>
<div>
 <button @click="show = !show">click4</button>
 <!-- https://daneden.github.io/animate.css/ -->
 <!-- 用这个网址样式，样式里必须用 animated -->
 <!-- bounceInRight rubberBand 网址里所选样式 -->
 <transition>
 enter-active-class="animated bounceInRight"
 leave-active-class="animated rubberBand"
 </transition>
 <div class="box" v-show="show">i am ok</div>
</div>
</template>
<script>
export default {
 data() {
 return {
 show: true,
 };
 },
}
</script>
<style scoped>
.box {
 margin-top: 30px;
 width: 150px;
 height: 150px;
 border: 1px solid red;
 text-align: center;
 line-height: 150px;
}
</style>
```

## 同时使用过渡和动画

可使用 `type` 属性，来声明需要 Vue 监听的类型，`type`值可为 `animation` 或 `transition`。

当不设置`type`时，默认会取 `transitioned` 和 `animationended` 两者更长的为结束时刻。如果设置，就会取设置类型的时间为准。

举例（组件）：

```

<template>
<div>
 <button @click="show = !show">click4</button>
 <!-- 添加type 过渡时间只有type的类型决定 -->
 <!-- 没写type 就会按时间更长的时间为准 -->
 <transition
 type="animation"
 enter-active-class="animated tada enter"
 leave-active-class="animated tada leave"
 >
 <div class="box" v-show="show">i am ok</div>
 </transition>
</div>
</template>
<script>
export default {
 data() {
 return {
 show: true,
 };
 },
</script>
<style scoped>
.box {
 margin-top: 30px;
 width: 150px;
 height: 150px;
 border: 1px solid red;
 text-align: center;
 line-height: 150px;
}
.v-enter,
.v-leave-to {
 opacity: 0;
}
.leave {
 transition: all 1s;
}
.enter {
 transition: all 3s;
}
.v-enter-to,
.v-leave {
 opacity: 1;
}
</style>

```

## 显性的过渡时间

在一些情况下，Vue可以自动得出过渡效果的完成时机，从而对dom进行处理。

但是有些时候，我们会设置一系列的过渡效果，例如嵌套元素也有过渡动效，其过渡效果的时间长于父元素。此时我们可以设置duration属性，定制一个显性的过渡持续时间（以毫秒记）：

```
<transition :duration="1000">...</transition>
```

也可以定制进入和移出的持续时间：

```
<transition :duration="{ enter: 500, leave: 800 }">...</transition>
```

举例（组件）：应用

```

<template>
<div>
 <button @click="show = !show">click4</button>
 <!-- duration 过渡时间, 如果有了它, 其他定义时间就不好使 -->
 <transition
 :duration="1000"
 enter-active-class="animated tada enter"
 leave-active-class="animated tada leave"
 >
 <div class="box" v-show="show">i am ok</div>
 </transition>
</div>
</template>
<script>
export default {
 data() {
 return {
 show: true,
 };
 },
}
</script>
<style scoped>
.box {
 margin-top: 30px;
 width: 150px;
 height: 150px;
 border: 1px solid red;
 text-align: center;
 line-height: 150px;
}
.v-enter,
.v-leave-to {
 opacity: 0;
}

.leave {
 transition: all 13s;
}
.enter {
 transition: all 10s;
}
.v-enter-to,
.v-leave {
 opacity: 1;
}
</style>

```

:duration也可以在父级定义, 以防父级动画结束后子级动画没完成, 在父级定义子级动画完成时间来完成子级定义的动画。

举例 (组件) : 应用

```

<template>
<div>
 <button @click="show = !show">click4</button>
 <!-- duration 过渡时间, -->
 <transition :duration="{ enter: 2000, leave: 6000 }">
 <div class="box" v-show="show">
 <transition name="span">
 hello world
 </transition>
 </div>
 </transition>
</div>
</template>
<script>
export default {
 data() {
 return {
 show: true,
 };
 },
}
</script>
<style scoped>
.box {
 margin-top: 30px;
 width: 150px;
 height: 150px;
 border: 1px solid red;
 text-align: center;
 line-height: 150px;
}
.v-enter,
.v-leave-to {
 opacity: 0;
}

.v-leave-active {
 transition: all 13s;
}
.v-enter-active {
 transition: all 10s;
}
.v-enter-to,
.v-leave {
 opacity: 1;
}
.span-enter,
.span-leave-to {
 font-size: 12px;
}

.span-enter-active,
.span-leave-active {
 transition: all 2s;
}
.span-enter-to,
.span-leave {
 font-size: 20px;
}
</style>

```

## 初始渲染的过渡

可以通过 `appear` attribute 设置节点在初始渲染的过渡。

和进入/离开过渡一样，同样也可以自定义 CSS 类名。如：

```

appear-class="appear-enter"
appear-active-class="appear-enter-active"
appear-to-class="appear-enter-to"

```

但是用自定义类名，需在类里添加 `appear`，才可以在初始渲染时产生过渡效果

举例（组件）：

```
<template>
<div>
 <button @click="show = !show">click</button>
 <!-- appear 刷新初始化页面会有过渡 -->
 <!-- appear-active-class 如果有对应类写在这里，但是也是要加 appear-->
 <transition appear appear-active-class="animated swing">
 <div class="box" v-if="show">i am ok</div>
 </transition>
</div>
</template>
<script>
export default {
 data() {
 return {
 show: true,
 };
 },
</script>
<style scoped>
.box {
 margin-top: 30px;
 width: 150px;
 height: 150px;
 border: 1px solid red;
 text-align: center;
 line-height: 150px;
}

.v-enter {
 opacity: 0;
}

.v-enter-active {
 transition: opacity 1s;
}

.v-enter-to {
 opacity: 1;
}

.v-leave {
 opacity: 1;
}

.v-leave-active {
 transition: opacity 3s;
}

.v-leave-to {
 opacity: 0;
}
</style>
```

## JavaScript 钩子

可以在属性中声明 JavaScript 钩子：

```
<transition
 @before-enter="beforeEnter"
 @enter="enter"
 @after-enter="afterEnter"
 @enter-cancelled="enterCancelled"

 @before-leave="beforeLeave"
 @leave="leave"
 @after-leave="afterLeave"
 @leave-cancelled="leaveCancelled"
>
 <!-- ... -->
</transition>
```

- **before-enter** 动画入场前，可以在其中设置元素开始动画之前的起始样式，参数为el代表元素
- **enter** 动画入场中，可以在其中写动画，在其中写done()可以手动直接调用after-enter方法，第一个参数为el代表元素，第二个参数为done。  
done.canceled = true可以调用enter-cancelled方法来取消动画，但是很多时候不好使，一般不使用。
- **after-enter** 动画完成后
- **enter-cancelled** 取消动画

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。这也避免过渡过程中 CSS 的影响。

设置了 appear 特性的 transition 组件，也存在自定义 JavaScript 钩子：

```
<transition
 appear
 v-on:before-appear="customBeforeAppearHook"
 v-on:appear="customAppearHook"
 v-on:after-appear="customAfterAppearHook"
 v-on:appear-cancelled="customAppearCancelledHook"
>
 <!-- ... -->
</transition>
```

举例（组件）：

```

<template>
<div>
 <button @click="show = !show">click</button>
 <!-- :css="false" 可以直接跳过css -->
 <transition
 :css="false"
 @before-enter="beforeEnter"
 @enter="enter"
 @after-enter="afterEnter"
 @enter-cancelled="enterCancelled"
 @before-leave="beforeLeave"
 @leave="leave"
 @after-leave="afterLeave"
 @leave-cancelled="leaveCancelled"
 >
 <div class="box" v-if="show">i am ok</div>
 </transition>
</div>
</template>
<script>
export default {
 data() {
 return {
 show: true,
 };
 },
 methods: {
 beforeEnter(el) {
 el.style.opacity = 0;
 },
 enter(el, done) {
 //在整个动画结束后，下一个动画开始时触发访问enterCancelled方法，
 //但是 done.canceled所在方法里有异步函数就不会执行enterCancelled方法
 //一般不会使用这个方法，因为不太好使
 //done.canceled = true;
 setTimeout(() => {
 done(); //主动调用afterEnter
 }, 5000);
 },
 afterEnter(el) {
 console.log(el, "afterEnter");
 el.style.opacity = 1;
 },
 enterCancelled(el) {
 console.log(el, "enterCancelled");
 },
 beforeLeave(el) {
 el.style.opacity = 1;
 },
 leave(el, done) {
 setTimeout(() => {
 done(); //主动调用afterLeave
 }, 5000);
 },
 afterLeave(el) {
 console.log(el, "afterLeave");
 el.style.opacity = 0;
 },
 leaveCancelled(el) {
 console.log(el, "afterEnter");
 },
 },
};
</script>
<style scoped>
.box {
 margin-top: 30px;
 width: 150px;
 height: 150px;
 border: 1px solid red;
 text-align: center;
 line-height: 150px;
}

.v-enter {
 opacity: 0;
}

```

```
}

.v-enter-active {
 transition: opacity 1s;
}

.v-enter-to {
 opacity: 1;
}

.v-leave {
 opacity: 1;
}

.v-leave-active {
 transition: opacity 3s;
}

.v-leave-to {
 opacity: 0;
}
</style>
```

### 结合 Velocity.js

Velocity.js 官网地址: <http://velocityjs.org/>

安装方式: `npm install velocity-animate`

具有过渡效果

`Velocity(元素, {变化状态},{时间duration,完成状态 (可填) })`

举例 (组件) :

```

<template>
<div>
 <button @click="show = !show">click</button>
 <!-- :css="false" 可以直接跳过css -->
 <!-- @before-leave="beforeLeave"
 @leave="leave"
 @after-leave="afterLeave"
 @leave-cancelled="leaveCancelled" -->
 <transition
 :css="false"
 @before-enter="beforeEnter"
 @enter="enter"
 @after-enter="afterEnter"
 @enter-cancelled="enterCancelled"
 >
 <div class="box" v-if="show">i am ok</div>
 </transition>
</div>
</template>
<script>
export default {
 data() {
 return {
 show: true,
 };
 },
 methods: {
 beforeEnter(el) {
 el.style.opacity = 0;
 },
 enter(el, done) {
 // 具有过渡效果 Velocity(元素, {变化状态}, {时间duration,完成状态 (可填) })
 Velocity(el, { opacity: 1 }, { duration: 1000 });
 Velocity(el, { rotateZ: 10 }, { duration: 300 });
 Velocity(el, { rotateZ: -10 }, { duration: 300 });
 // complete: done 这句可以在访问enter方法之后直接访问afterEnter方法
 Velocity(el, { rotateZ: 0 }, { duration: 300, complete: done });
 },
 afterEnter(el) {
 console.log("afterEnter");
 },
 enterCancelled(el) {},
 beforeLeave(el) {},
 leave(el, done) {},
 afterLeave(el) {},
 leaveCancelled(el) {},
 },
};
</script>
<style scoped>
.box {
 margin-top: 30px;
 width: 150px;
 height: 150px;
 border: 1px solid red;
 text-align: center;
 line-height: 150px;
}

.v-enter {
 opacity: 0;
}

.v-enter-active {
 transition: opacity 1s;
}

.v-enter-to {
 opacity: 1;
}

.v-leave {
 opacity: 1;
}

.v-leave-active {

```

```

 transition: opacity 3s;
}

.v-leave-to {
 opacity: 0;
}

```

## 过渡\_多元素过渡（切换展示多元素）

当切换展示的元素标签名相同时，需要给每一个元素设置不同的key值，否则Vue为了效率只会替换相同标签内部的内容。

```

<transition>
 <div v-if="show" key="world">hello world</div>
 <div v-else key="shanshan">hello shanshan</div>
</transition>

```

举例（组件）：

```

<template>
 <div>
 <button @click="show = !show">click</button>
 <!-- 多个div 过渡 -->
 <transition>
 <div v-if="show" key="world">hello world</div>
 <div else key="shanshan">hello shanshan</div>
 </transition>
 </div>
</template>
<script>
export default {
 data() {
 return { show: true };
 },
};
</script>
<style scoped>
div {
 margin-top: 15px;
}
.v-enter,
.v-leave-to {
 opacity: 0;
}
.v-enter-active,
.v-leave-active {
 transition: all 0.5s;
}
.v-enter-to,
.v-leave {
 opacity: 1;
}
</style>

```

在一些场景中，可以通过给同一个元素的key值设置不同的状态来替代 v-if 和 v-else。如：

```

<transition>
 <div :key="keyName">hello {{ keyName }}</div>
</transition>

```

```
keyName: 'world',
```

举例（组件）：

```
<template>
<div>
 <button @click="handle()">click</button>
 <!-- 多个div 过渡 -->
 <transition>
 <div :key="keyName">hello {{ keyName }}</div>
 </transition>
</div>
</template>
<script>
export default {
 data() {
 return { keyName: "world" };
 },
 methods: {
 handle() {
 this.keyName = this.keyName === "world" ? "shanshan" : "world";
 },
 },
};
</script>
<style scoped>
div {
 margin-top: 15px;
}
.v-enter,
.v-leave-to {
 opacity: 0;
}
.v-enter-active,
.v-leave-active {
 transition: all 0.5s;
}
.v-enter-to,
.v-leave {
 opacity: 1;
}
</style>
```

## 过渡模式

Vue提供一个一个 mode 特性，可以给多个元素过渡应用不同的模式，mode 的值可为：

- in-out: 新元素先进行过渡，完成之后当前元素过渡离开。
- out-in: 当前元素先进行过渡，完成之后新元素过渡进入。

举例（组件）：

```
<template>
<div>
 <button @click="handle()">click</button>
 <!-- mode='out-in' 当前元素先进行过渡，完成之后新元素过渡进入 -->
 <transition mode="out-in">
 <div :key="keyName">hello {{ keyName }}</div>
 </transition>
</div>
</template>
<script>
export default {
 data() {
 return { keyName: "world" };
 },
 methods: {
 handle() {
 this.keyName = this.keyName === "world" ? "shanshan" : "world";
 },
 },
};
</script>
<style scoped>
div {
 margin-top: 15px;
}
.v-enter,
.v-leave-to {
 opacity: 0;
}
.v-enter-active,
.v-leave-active {
 transition: all 0.5s;
}
.v-enter-to,
.v-leave {
 opacity: 1;
}
</style>
```

## 多组件过渡

我们可以使用动态组件切换展示不同的组件。

举例（组件）：

```

<template>
<div>
 <button @click="handle()">click</button>
 <!-- 动态组件 -->
 <transition mode="out-in">
 <component :is="cmpName"></component>
 </transition>
</div>
</template>
<script>
import Demo1 from "./BaseDemo5-1";
import Demo2 from "./BaseDemo5-2";
export default {
 components: {
 Demo1,
 Demo2,
 },
 data() {
 return { cmpName: "demo1" };
 },
 methods: {
 handle() {
 this.cmpName = this.cmpName === "demo1" ? "demo2" : "demo1";
 },
 },
};
</script>
<style scoped>
div {
 margin-top: 15px;
}
.v-enter,
.v-leave-to {
 opacity: 0;
}
.v-enter-active,
.v-leave-active {
 transition: all 0.5s;
}
.v-enter-to,
.v-leave {
 opacity: 1;
}
</style>

```

## 过渡\_列表过渡

当想要给一个列表添加过渡动效时，我们可以使用 `<transition-group>` 组件。

该组件的特点：

- 不同于 `<transition>`，它会以一个真实元素呈现：默认为一个 `<span>` 来包裹 `transition-group` 内部元素。你也可以通过 `tag attribute` 更换为其他元素。
- 过渡模式不可用，因为我们不再相互切换特有的元素。
- 内部元素 总是需要 提供唯一的 `key` 属性值。
- CSS 过渡的类将会应用在内部的元素中，而不是这个组/容器本身。

举例（组件）：

```

<template>
<div class="demo">
 <button @click="show = !show">click</button>
 <!-- transition-group 包裹多元素过渡 一定需要key-->
 <!-- tag 可用来定义包裹元素标签 -->
 <transition-group tag="div">
 <div key="world" v-if="show">hello wrold</div>
 <div key="shanshan" v-if="show">hello shanshan</div>
 </transition-group>
</div>
</template>

<script>
export default {
 data() {
 return {
 show: true,
 };
 },
}
</script>

<style scoped>
.v-enter,
.v-leave-to {
 opacity: 0;
}

.v-enter-active,
.v-leave-active {
 transition: all 0.3s;
}

.v-leave,
.v-enter-to {
 opacity: 1;
}
</style>

```

## 列表的排序过渡

<transition-group> 组件提供了一个新的特性: v-move, 它会在元素改变定位的过程中应用。

如何使用? 通过类名即可设置: .v-move {}。

当给 <transition-group> 设置 name 特性时, 例如name为fade, 则 v-move 在使用时, 需要替换为 fade-move。

注意: 当移除一个列表元素时, 需要将移除的元素脱离文档流, 否则, 要溢出的元素在移除过渡中一直处于文档流中, 会影响到后面元素的move过渡效果。

内部的实现: Vue 使用了一个叫 FLIP 简单的动画队列, 使用 transforms 将元素从之前的位置平滑过渡新的位置。

需要注意的是使用 FLIP 过渡的元素不能设置为 display: inline 。作为替代方案, 可以设置为 display: inline-block 或者放置于 flex 中。

举例 (组件) :

```

<template>
 <div class="demo">
 <button @click="handleAdd">添加</button>
 <button @click="handleRemove">移除</button>
 <button @click="handleShuffle">洗牌</button>

 <transition-group tag="ul">
 <li v-for="item in lists" :key="item">{{ item }}
 </transition-group>
 </div>
</template>

<script>
export default {
 data() {
 return {
 lists: [1, 2, 3, 4, 5, 6, 7, 8, 9, 10],
 nextNum: 11,
 };
 },
 methods: {
 handleAdd() {
 const index = Math.floor(Math.random() * this.lists.length);
 this.lists.splice(index, 0, this.nextNum++);
 },
 handleRemove() {
 const index = Math.floor(Math.random() * this.lists.length);
 this.lists.splice(index, 1);
 },
 handleShuffle() {
 this.lists.sort(() => Math.random() - 0.5);
 },
 },
};
</script>

<style scoped>
button {
 margin-bottom: 10px;
 margin-right: 10px;
}

ul,
li {
 padding: 0;
 margin: 0;
}

li {
 list-style: none;
 /* 这要用inline就不会不好使 */
 display: inline-block;
 margin-right: 10px;
}

.v-enter,
.v-leave-to {
 opacity: 0;
 transform: translateY(30px);
}

.v-enter-active,
.v-leave-active {
 transition: all 0.3s;
}

.v-leave,
.v-enter-to {
 opacity: 1;
 transform: translateY(0px);
}

/* 洗牌时样式
 * 因为v-enter-active v-leave-active用的是transform
 * 所以洗牌时用 transform

```

```
 */
.v-move {
 transition: transform 0.3s;
}
</style>
```

## 列表的交错过渡

如果要给列表中的元素，应用更丰富的过渡效果，可以配合JavaScript钩子。

举例（组件）：

```
<template>
<div class="demo">
 <input type="text" v-model="query">
 <transition-group
 tag="ul"
 @before-enter="beforeEnter"
 @enter="enter"
 @leave="leave"
 >
 <li
 v-for="item in computedLists"
 :key="item.name"
 >{{ item.name }}
 </transition-group>
</div>
</template>

<script>
export default {
 data () {
 return {
 query: '',
 lists: [
 { name: 'shanshan'},
 { name: 'jicheng'},
 { name: 'chensitong'},
 { name: 'dengxuming'},
]
 }
 },
 computed: {
 computedLists () {
 return this.lists.filter(item => item.name.includes(this.query));
 },
 },
 methods: {
 beforeEnter (el) {
 el.style.opacity = 0;
 el.style.height = 0;
 },
 enter (el, done) {
 Velocity(el, { opacity: 1, height: '24px' }, { duration: 300, complete: done })
 },
 leave (el, done) {
 Velocity(el, { opacity: 0, height: '0px' }, { duration: 300, complete: done })
 }
 },
}
</script>

<style scoped>
li {
 height: 24px;
}
</style>
```

## 过渡\_复用过渡

过渡可以通过 Vue 的组件系统实现复用。要创建一个可复用过渡组件，你需要做的就是将 或者 作为根组件，然后将任何子组件放置在其中就可以了。

注意：当使用函数式组件复用过渡时，不要设置css作用域（style 里的scoped需要去除，如果不去除过渡就会不好使）

举例（组件）：

主 组件

```
<template>
<div>
 <button @click="show = !show">点我</button>
 <fu-yong>
 <div v-if="show">hello world</div>
 </fu-yong>
</div>
</template>
<script>
//组件 过渡复用
//import FuYong from "./FuYong";
//函数式组件 过渡复用
//import FuYong from "./FuyongHanshu";
export default {
 components: {
 FuYong,
 },
 data() {
 return {
 show: true,
 };
 },
}
</script>
<style scoped>
button {
 margin-bottom: 30px;
}
</style>
```

调用组件<sup>①</sup>

```
<template>
<div>
 <transition>
 <slot></slot>
 </transition>
</div>
</template>
<script>
export default {
 name: "FuYong",
};
</script>
<style scoped>
.v-enter,
.v-leave-to {
 opacity: 0;
}
.v-enter-active,
.v-leave-active {
 transition: all 3s;
}
.v-enter-to,
.v-leave {
 opacity: 1;
}
</style>
```

调用组件<sup>②</sup>

```

<script>
//函数式组件 复用
export default {
 name: "FuyongHanshu",
 functional: true,
 render(h, context) {
 const { slots } = context;
 return (
 <div>
 <transition name="single">{slots().default}</transition>
 </div>
);
 },
}
</script>
<style >
/* style 里的scoped使用函数式组件需要去除，如果不去除过渡就会不好使 */
.single-enter,
.single-leave-to {
 opacity: 0;
}
.single-enter-active,
.single-leave-active {
 transition: all 3s;
}
.single-enter-to,
.single-leave {
 opacity: 1;
}
</style>

```

## 组件\_异步组件

在项目中，有些组件不会在第一次进入首屏时加载，而是当执行了某些操作时，才会加载进来，所以此时，我们可以将该组件设置成异步加载，什么时候用，什么时候再加载进来，以达到提升首屏性能的目的。

使用方法：

```

components: {
 AsyncCmp: () => import (url);
}

```

举例（组件）：

```

<template>
 <div>
 <button @click="show = !show">点我</button>
 <async-demo v-if="show"> </async-demo>
 </div>
</template>
<script>
export default {
 components: {
 AsyncDemo: () => import("./BaseDemo2"),
 },
 data() {
 return {
 show: false,
 };
 },
}
</script>
<style scoped>
button {
 margin-bottom: 30px;
}
</style>

```

将多个需要同时加载的组件合并到一个文件中，文件名为async，这样效率会提高：

```
components: {
 AsyncCmp1: () => import(/* webpackChunkName: "async" */ 'url'),
 AsyncCmp2: () => import(/* webpackChunkName: "async" */ 'url'),
}
```

举例（组件）：

```
<template>
 <div>
 <button @click="show = !show">点我</button>
 <async-demo2 v-if="show"> </async-demo2>
 <async-demo3 v-if="show"> </async-demo3>
 </div>
</template>
<script>
export default {
 components: {
 AsyncDemo2: () => import(/* webpackChunkName: "async" */ "./BaseDemo2"),
 AsyncDemo3: () => import(/* webpackChunkName: "async" */ "./BaseDemo3"),
 },
 data() {
 return {
 show: false,
 };
 },
}
</script>
<style scoped>
button {
 margin-bottom: 30px;
}
```

```
<link href="/js/0.js" rel="prefetch">
<link href="/js/app.js" rel="preload" as="script">
```

异步加载的文件，会在link标签上设置 el="prefetch"。浏览器会在空闲时间内下载对应的资源，使用时可以直接从缓存中获取。与之对应的 el="preload"，会及时下载对应的资源。

( <script type="text/javascript" src="/js/app.js"></script> 这个js里写的都是我们写的东西。 <script type="text/javascript" src="/js/chunk-vendors.js"></script> 这里面写的是我们引入的组件)

## VueRouter\_基础

### 什么是路由？

路由是根据不同的url地址展现不同的内容或页面。

早期的路由都是后端直接根据url来重载页面实现的，即后端控制路由。

后来页面越来越复杂，服务器压力越来越大，随着ajax（异步刷新技术）的出现，页面的实现非重载就能刷新数据，让前端也可以控制url自行管理，前端路由由此而生。

### 什么时候使用前端路由？

前端路由更多用在单页应用上，也就是SPA(Single Page Web Application)，在单页面应用中，大部分页面结果不变，只改变部分内容的使用。

## 安装路由

安装： `npm install vue-router`。

`vue/cli` 这个命令集中有一条是 `@` 代表路径从 `src` 开始算

举例： 路径为 `src/components/`

```
import AddStudent from "@/components/AddStudent";
```

## 使用路由

## JavaScript

### 1. 引入路由

```
import VueRouter from 'vue-router';
```

### 2. 使用路由

```
Vue.use(VueRouter);
```

因为这句：我们才能使用 `<router-link>` 和 `<router-view>` 和 `route` 和 `router`  
这句其实用的就是js中的`install`函数，我们通常用一个js文件来写

```
export default function install(Vue) {

}
```

### 3. 定义路由组件

```
// 可以从其他文件 import 进来
const Foo = { template: '<div>foo</div>' }
const Bar = { template: '<div>bar</div>' }
或是
import xxx1 from './路径/xxx1.vue'
import xxx2 from './路径/xxx2.vue'
```

### 4. 定义路由

```
// 每个路由应该映射一个组件
const routes = [
 { path: '/foo', component: Foo },
 { path: '/bar', component: Bar }
]
```

### 5. 创建 router 实例，然后传 routes 配置

```
const router = new VueRouter({
 routes
})
```

### 6. 创建和挂载根实例

```
const app = new Vue({
 router
}).$mount('#app')
```

## html

```
<div id="app">
 <h1>Hello App!</h1>
 <p>
 <!-- 使用 router-link 组件来导航。-->
 <!-- 通过传入 `to` 属性指定链接。to='对应的path'-->
 <!-- <router-link> 默认会被渲染成一个 `<a>` 标签 -->
 <!-- 如果不加tag router-link则会转化为<a> 加tag则会转化为tag的对应标签 -->
 <router-link to="/foo" tag="div">Go to Foo</router-link>
 <router-link to="/bar">Go to Bar</router-link>
 </p>
 <!-- 路由出口 -->
 <!-- 路由匹配到的组件将渲染在这里 -->
 <router-view></router-view>
</div>
```

## router-link class

- `router-link-exact-active` 当前展示路径完全匹配组件`to`属性的值(就是当前选中的`class`值，我们一般看那个页面被选中，就看这个值)，但是可能会有这样一种情况，我们的菜单下有子菜单，那么这是你要配类`router-link-exact-active`就会出现选择子菜单时，对应所属菜单不被选中，这时，我们就会用类`router-link-active`，就可以了。
- `router-link-active` 当前展示路径包含`to`属性的值

更改`class`名，通过`VueRouter`修改选中的`class`值

```
VueRouter({
 linkActiveClass: 'link-active',
 linkExactActiveClass: 'link-exact-active',
})
```

## hash模式

`vue-router` 默认 hash 模式 — 使用 URL 的 hash 来模拟一个完整的 URL，于是当 URL 改变时，页面不会重新加载。

URL类似于这样：<http://localhost:8081/demo2#/demo2>

原理：浏览器hash模式

```
location.hash = '/about'
```

## history 模式

原理：浏览器history模式

```
history.pushState(null, null, '/about');
```

如果不想要很丑的 hash，我们可以用路由的 history 模式，这种模式充分利用 `history.pushState` API 来完成 URL 跳转而无须重新加载页面。在路由配置中设置：

```
VueRouter({
 mode: 'history',
})
```

当你使用 history 模式时，URL 就像正常的 url，例如 <http://yoursite.com/user/id>，也好看！

不过这种模式要玩好，还需要后台配置支持。因为我们的应用是个单页客户端应用，如果后台没有正确的配置，当用户在浏览器直接访问 <http://oursite.com/user/id> 就会返回 404，这就不好看了。

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。

举例：将路由放在一个`router.js`里，由`main.js`引用

```
router.js
```

```

import VueRouter from "vue-router";
import Vue from 'vue'

Vue.use(VueRouter);

const routes = [
 {
 path: "/",
 component: () => import("./components/BaseDemo1.vue")
 },
 {
 path: "/demo2",
 component: () => import("./components/BaseDemo2.vue")
 },
 {
 path: "/demo3",
 component: () => import("./components/BaseDemo3.vue")
 },
];
;

const router = new VueRouter({
 //加载 路径 组件
 routes,
 //网络url 不用哈希，用history，普遍来讲，都用这个
 mode: "history",
 //类名
 linkActiveClass: "link-active",
 //选中完全匹配的类名
 linkExactActiveClass: "link-exact-active",
});
;

export default router;

```

main.js

```

import Vue from 'vue'
import App from './App.vue'
//引入写的路由文件
import router from './router'

Vue.config.productionTip = false

new Vue({
 render: h => h(App),
 router
}).$mount('#app')

```

## VueRouter\_命名路由-嵌套路由-重定向-别名

### 命名路由

可以通过一个名称标识一个路由，这样在某些时候会显得更方便一些，特别是在链接一个路由，或者是执行一些跳转时，可以在创建Router实例时，在routes配置中给某个路由设置名称：

```

routes = [
 {
 path: '/activity/personal',
 name: 'personal',
 component: Personal,
 }
];

```

要链接到一个命名路由，可以给 `router-link` 的 `to` 属性传一个对象：

```
<router-link :to="{ name: 'personal' }">个人中心</router-link>
```

### 嵌套路由

一个被 `router-view` 渲染的组件想要包含自己的嵌套 `router-view` 时，可以使用嵌套路由，如：

```
{
 path: '/activity',
 component: () => import('./views/Activity'),
 children: [
 {
 path: '/activity/academic',
 name: 'academic',
 component: () => import('./views/Academic'),
 },
 {
 path: '/activity/personal',
 name: 'personal',
 component: () => import('./views/Personal'),
 },
 {
 path: '/activity/download',
 name: 'download',
 component: () => import('./views/Download'),
 }
],
}
```

经过这样的设置，在 `Activity` 组件中就可以使用 `router-view` 了。

子路由的 `path` 可以简写：

```
path: 'personal'
```

这样会自动将父路由的路径，拼接在子路由前，最终结果为：`/activity/personal`。

举例：

```
const routes = [
 {
 path: '/',
 component: () => import('./components/BaseDemo1')
 },
 {
 path: '/demo2',
 component: () => import('./components/BaseDemo2')
 },
 {
 path: '/demo3',
 component: () => import('./components/BaseDemo3')
 },
 {
 path: '/demo4',
 component: () => import('./components/BaseDemo4'),
 children: [
 {
 name: '4-1',
 // 简写 完整的是 /demo4/BaseDemo4-1
 path: "BaseDemo4-1",
 component: () => import('./components/BaseDemo4-1')
 },
 {
 name: '4-2',
 path: "BaseDemo4-2",
 component: () => import('./components/BaseDemo4-2')
 },
 {
 name: '4-3',
 path: "BaseDemo4-3",
 component: () => import('./components/BaseDemo4-3')
 }
]
}
```

当访问 `/activity` 下的其他路径时，并不会渲染出来任何东西，如果想要渲染点什么，可以提供一个空路由：

```
{
 path: '/activity',
 children: [
 {
 path: '',
 component: () => import('./views/Academic'),
 },
],
}
```

# 重定向

重定向也是通过 routes 配置来完成，下面例子是从 /a 重定向到 /b，b为对应的path

```
const router = new VueRouter({
 routes: [
 { path: '/a', redirect: '/b' }
]
})
```

举例：

```
const routes = [
 //为了在首页直接可以显示这个组件
 path: '/',
 //重定向 redirect:'对应的path'
 redirect: '/demo1'
}, {
 //为了让linkActiveClass类在选中别的类时不匹配上这个组件,所以给了这个组件除了'/'以外的path
 path: '/demo1',
 component: () => import('./components/BaseDemo1')
}]
const router = new VueRouter({
 routes,
 mode: 'history',
 //类名
 linkActiveClass: "link-active",
 //选中完全匹配的类名
 linkExactActiveClass: "link-exact-active"
})
```

重定向的目标也可以是一个命名的路由：

```
const router = new VueRouter({
 routes: [
 { path: '/a', redirect: { name: 'foo' } }
]
})
```

甚至是一个方法，动态返回重定向目标：

```
const router = new VueRouter({
 routes: [
 { path: '/a', redirect: to => {
 // 方法接收 目标路由 作为参数
 // return 重定向的 字符串路径/路径对象
 }}
]
})
```

重定向与对应children中空路径：如果对应children配置空路径，重定向则不会生效。

比如下面的例子：

```

const routes = [
{
 path: '/activity',
 component: () => import('../views/Activity'),
 redirect(to) {
 return {
 name: 'academic',
 }
 },
 children: [
 {
 path: '',
 component: () => import('../views/Academic'),
 },
],
},
];
export default new VueRouter({
 mode: 'history',
 routes,
});

```

## 别名

“重定向”的意思是，当用户访问 /a 时，URL 将会被替换成 /b，然后匹配路由为 /b，那么“别名”又是什么呢？

/a 的别名是 /b，意味着，当用户访问 /b 时，URL 会保持为 /b，但是路由匹配则为 /a，就像用户访问 /a 一样。

上面对应的路由配置为：

```

const router = new VueRouter({
 routes: [
 { path: '/a', component: A, alias: '/b' }
]
})

```

## VueRouter\_编程式的导航

通过在 Vue 根实例的 router 配置传入 router 实例，\$router、\$route 两个属性会被注入到每个子组件。

### \$router

路由实例对象。

除了使用 <router-link> 创建 a 标签来定义导航链接，我们还可以借助 router 的实例方法，通过编写代码来实现。

### \$router.push

想要导航到不同的 URL，则使用 router.push 方法。这个方法会向 history 栈添加一个新的记录，所以，当用户点击浏览器后退按钮时，则回到之前的 URL。

当你点击 <router-link> 时，这个方法会在内部调用，所以说，点击 <router-link :to="..."> 等同于调用 \$router.push(...)

声明式	编程式
<router-link :to="...">	this.\$router.push(...)

该方法的参数可以是一个字符串路径，或者一个描述地址的对象。例如：

```
// 字符串 值为path值
this.$router.push('home')

// 对象
this.$router.push({ path: 'home' })

// 命名的路由
this.$router.push({ name: 'user' })
```

## \$router.replace

跟 `router.push` 很像，唯一的不同就是，它不会向 `history` 添加新记录，而是替换掉当前的 `history` 记录。

声明式	编程式
<router-link :to="..." replace>	this.\$router.replace(...)

```
// 字符串 值为path值
this.$router.replace('home')

// 对象
this.$router.replace({ path: 'home' })

// 命名的路由
this.$router.replace({ name: 'user' })
```

举例：

```
export default {
 name: "App",
 components: {},
 methods: {
 handleClick() {
 //console.log(this.$router);
 this.$router.push("/demo1");
 //this.$router.replace("/demo1");
 },
 },
};
```

## \$router.go(n)

这个方法的参数是一个整数，意思是在 `history` 记录中向前或者后退多少步，类似 `window.history.go(n)`。

```
//相当于刷新页面
this.$router.go(0)

// 在浏览器记录中前进一步，等同于 history.forward()
this.$router.go(1)

// 后退一步记录，等同于 history.back()
this.$router.go(-1)

// 前进 3 步记录
this.$router.go(3)

// 如果 history 记录不够用，那就默默地失败呗
this.$router.go(-100)
this.$router.go(100)
```

## \$route

只读，路由信息对象。

## \$route.path

字符串，对当前路由的路径，总是解析为绝对路径，如 `"/foo/bar"`。

## \$route.params

一个 key/value 对象，包含了动态片段和全匹配片段，如果没有路由参数，就是一个空对象。

```
<router-link :to="{ name: 'question', params: { id: question.id } }">
 {{ question.title }}
</router-link>
```

## \$route.query

一个 key/value 对象，表示 URL 查询参数。例如，对于路径 /foo?user=1，则有 \$route.query.user == 1，如果没有查询参数，则是个空对象。

## \$route.hash

路由的 hash 值（带 #），如果没有 hash 值，则为空字符串。

## \$route fullPath

完成解析后的 URL，包含查询参数和 hash 的完整路径。

## \$route.matched

一个数组，包含当前路由的所有嵌套路经片段的路由记录。路由记录就是 routes 配置数组中的对象副本（还有在 children 数组）。

```
js const router = new VueRouter({ routes: [// 下面的对象就是路由记录 { path: '/foo', component: Foo, children: [// 这也是个路由记录 { path:
```

当 URL 为 /foo/bar，\$route.matched 将会是一个包含从上到下的所有对象（副本）。

## \$route.name

当前路由的名称，如果有的话

## \$route.redirectedFrom

如果存在重定向，即为重定向来源的路由的名字。

# VueRouter\_ 动态路由匹配

当我们需要把某种模式匹配到的所有路由，全都映射到同一个组件。例如，我们有一个 User 组件，对于所有 ID 各不相同的用户，都要使用这个组件来渲染。那么，我们可以在 vue-router 的路由路径中使用“动态路径参数”来达到这个效果：

```
const User = {
 template: '<div>User</div>'
}

const router = new VueRouter({
 routes: [
 // 动态路径参数 以冒号开头
 { path: '/user/:id', component: User }
]
})
```

经过这样的设置，像 /user/foo 和 /user/bar 都将映射到相同的路由。

一个“路径参数”使用冒号 : 标记。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。

# VueRouter\_ 命名视图-路由组件传参

## 命名视图

想同时展示多个视图时，并且每个视图展示不同的组件时，可以使用命名视图。

可以在界面中拥有多个单独命名的视图，而不是只有一个单独的出口。如果 `router-view` 没有设置名字，那么默认为 `default`。

```
<router-view class="view one"></router-view>
<router-view class="view two" name="a"></router-view>
<router-view class="view three" name="b"></router-view>
```

一个视图使用一个组件渲染，因此对于同个路由，多个视图就需要多个组件。确保正确使用 `components` 配置（带上 `s`）：

```
const router = new VueRouter({
 routes: [
 {
 path: '/',
 components: {
 default: Foo,
 a: Bar,
 b: Baz
 }
 }
]
})
```

举例：

```
<div class="view">
 <router-view></router-view>
 <router-view name="minming"></router-view>
</div>

const routes = [{
 path: '/demo2',
 components: {
 //命名为minming的路由
 minming: () => import('./components/BaseDemo3'),
 //如果vue文件上没写name 则默认是default
 default: () => import('./components/BaseDemo2')
 }
}
]

const router = new VueRouter({
 routes,
 mode: 'history',
 //类名
 linkActiveClass: "link-active",
 //选中完全匹配的类名
 linkExactActiveClass: "link-exact-active"
})
```

## 路由组件传参

在组件中使用 `$route` 会使之与其对应路由形成高度耦合，从而使组件只能在某些特定的 URL 上使用，限制了其灵活性。

使用 `props` 将组件和路由解耦。

## 布尔模式

如果 `props` 被设置为 `true`, `route.params` 将会被设置为组件属性。

举例：

```
route.js
```

```

const routes = [
 {
 name: 'question',
 path: '/demo4/question/:id',
 //设置为true
 props: true,
 component: () => import('./components/Question')
 }
]

const router = new VueRouter({
 routes,
 mode: 'history',
 //类名
 linkActiveClass: "link-active",
 //选中完全匹配的类名
 linkExactActiveClass: "link-exact-active"
})

```

## Question.vue

访问 <http://localhost:8081/demo4/question/90878976>, 在这id为90878976

```

export default {
 //接收路由参数, 如果不接收id,id则不会有值
 props: ["id"],
 mounted() {
 //这里 会接收从$route传来的id值
 console.log(this.id);
 },
}

```

## 对象模式

如果 `props` 是一个对象, 它会被按原样设置为组件属性。当 `props` 是静态的时候有用。

```

const router = new VueRouter({
 routes: [
 {
 path: '/promotion/from-newsletter',
 component: Promotion,
 props: { id: 243232 }
 }
]
})

```

举例:

### route.js

```

const routes = [
 {
 name: 'question',
 path: '/demo4/question/:id',
 //设置id 因为每个路由都需要配置一个, 所以基本不用这种
 props: { id: 243232 },
 component: () => import('./components/Question')
 }
]

const router = new VueRouter({
 routes,
 mode: 'history',
 //类名
 linkActiveClass: "link-active",
 //选中完全匹配的类名
 linkExactActiveClass: "link-exact-active"
})

```

## Question.vue

```

export default {
 //接收路由参数，如果不接收id,id则不会有值
 props: ["id"],
 mounted() {
 //这里 会接收从$route传来的id值
 console.log(this.id);
 },
}

```

## 函数模式

你可以创建一个函数返回 `props`。函数的第一个参数是 `route`（即`$route`）。

```

const router = new VueRouter({
 routes: [
 { path: '/search', component: SearchUser, props: (route) => ({ query: route.query.q }) }
]
})

```

举例: `route.js`

```

const routes = [
 {
 //具体问题页面
 name: 'question',
 path: '/demo4/question/:id',
 //props为一函数 参数就是$route
 props: route => ({
 id: route.params.id,
 name: route.name
 }),
 component: () => import('./components/Question')
 }
]

const router = new VueRouter({
 routes,
 mode: 'history',
 //类名
 linkActiveClass: "link-active",
 //选中完全匹配的类名
 linkExactActiveClass: "link-exact-active"
})

```

使用: `Question.vue`

```

export default {
 //接收路由参数，如果不接收id name,id name则不会有值
 props: ["id", "name"],
 data() {
 return {
 question: null,
 };
 },
 mounted() {
 console.log(this.id);
 console.log(this.name);
 }
};

```

如果有别的vue文件 (`BaseDemo2.vue`) 想调用需要路由信息的vue文件(`Question.vue`)时  
`BaseDemo2.vue`

```

<!-- 传参 id name -->
<base-question id="90878976" name="question"></base-question>

```

`Question.vue`

```
export default {
 //接收路由参数，如果不接收id name,id name则不会有值
 props: ["id", "name"],
 data() {
 return {
 question: null,
 };
 },
 mounted() {
 console.log(this.id);
 console.log(this.name);
 }
};
```

## VueRouter\_导航守卫

导航：路由正在发生变化。

导航守卫主要用来通过跳转或取消的方式守卫导航。

导航守卫被分成三种：全局的、单个路由独享的、组件内的。

### 全局守卫

是指路由实例上直接操作的钩子函数，触发路由就会触发这些钩子函数。

#### 全局前置守卫 beforeEach

在路由跳转前触发，一般被用于登录验证。

```
const router = new VueRouter({ ... })

router.beforeEach((to, from, next) => {
 // ...
})
```

参数说明：

- to 目标路由对象
- from 即将要离开的路由对象
- next 三个参数中最重要的参数。
  - 必须调用next()，才能继续往下执行一个钩子，否则路由跳转会停止
  - 若要中断当前的导航，可以调用next(false)。
  - 可以使用next跳转到一个不同的地址。终端当前导航，进入一个新的导航。next参数值和\$router.push一致。
  - next(error)。2.4+，如果传入的参数是一个Error实例，则导航会被终止，且该错误会被传递给router.onError() 注册过的回调。

#### 全局解析守卫 beforeResolve

和beforeEach类似，路由跳转前触发。

和beforeEach的区别：在导航被确认之前，同时在所有组件内守卫和异步路由组件被解析之后，解析守卫就被调用。

```
const router = new VueRouter({ ... })

router.beforeResolve((to, from, next) => {
 // ...
})
```

#### 全局后置钩子 afterEach

和beforeEach相反，路由跳转完成后触发。

```
const router = new VueRouter({ ... })

router.afterEach((to, from) => {
 // ...
})
```

## 路由独享守卫

是指在单个路由配置的时候也可以设置的钩子函数。

### beforeEnter

和beforeEach完全相同，如果都设置则在beforeEach之后紧随执行。

```
const router = new VueRouter({
 routes: [
 {
 path: '/home',
 component: Home,
 beforeEnter: (to, from, next) => {
 // ...
 }
 }
]
})
```

## 组件内守卫（页面级守卫）

是指在组件内（component）执行的钩子函数，类似于组件内的生命周期，相当于为配置路由的组件添加的生命周期钩子函数。

### beforeRouteEnter

路由进入之前调用。

在该守卫内访问不到组件的实例，this值为undefined。在这个钩子函数中，可以通过传一个回调给next来访问组件实例。在导航被确认的时候执行回调，并且把组件实例作为回调方法的参数，可以在这个守卫中请求服务端获取数据，当成功获取并能进入路由时，调用next并在回调中通过vm访问组件实例进行赋值等操作，（next中函数的调用在mounted之后：为了确保能对组件实例的完整访问）。

```
beforeRouteEnter (to, from, next) {
 // 在渲染该组件的对应路由被 confirm 前调用
 // 不! 能! 获取组件实例 `this`
 // 因为当守卫执行前，组件实例还没被创建

 next(vm => {
 // 通过 `vm` 访问组件实例
 })
},
```

### beforeRouteUpdate

在当前路由改变时，并且该组件被复用时调用，可以通过this访问实例。

何时组件会被复用？

- 动态路由间互相跳转
- 路由query变更

```
beforeRouteUpdate (to, from, next) {
 // 在当前路由改变，但是该组件被复用时调用
 // 举例来说，对于一个带有动态参数的路径 /foo/:id，在 /foo/1 和 /foo/2 之间跳转的时候，
 // 由于会渲染同样的 Foo 组件，因此组件实例会被复用。而这个钩子就会在这个情况下被调用。
 // 可以访问组件实例 `this`
},
```

## beforeRouteLeave

导航离开该组件的对应路由时调用，可以访问组件实例this。

```
beforeRouteLeave (to, from, next) {
 // 导航离开该组件的对应路由时调用
 // 可以访问组件实例 `this`
}
```

举例(组件):

```
export default {
 beforeRouteUpdate(to, from, next) {
 console.log("beforeRouteUpdate question");
 next((vm) => {
 //vm相当于this
 console.log(vm);
 });
 },
 beforeRouteEnter(to, from, next) {
 console.log("beforeRouteEnter question");
 next();
 },
 beforeRouteLeave(to, from, next) {
 console.log("beforeRouteLeave question");
 next();
 },
};
```

## 完整的导航解析流程

1. 导航被触发。
2. 在失活的组件里调用离开守卫 组件守卫 beforeRouteLeave。
3. 调用全局的 beforeEach 守卫。
4. 在重用的组件里调用 beforeRouteUpdate 守卫 (2.2+)。
5. 在路由配置里调用 beforeEnter。
6. 解析异步路由组件。
7. 在被激活的组件里调用 beforeRouteEnter。
8. 调用全局的 beforeResolve 守卫 (2.5+)。
9. 导航被确认。
10. 调用全局的 afterEach 钩子。
11. 触发 DOM 更新 (现在已经是mounted之后)。
12. 用创建好的实例调用 beforeRouteEnter 守卫中传给 next 的回调函数。

## VueRouter\_路由元信息

定义路由的时候可以配置 meta 字段，用于自定义一些信息。路由的对应vue文件里的\$route存有meta值

```
const router = new VueRouter({
 routes: [
 {
 path: '/foo',
 component: Foo,
 children: [
 {
 path: 'bar',
 component: Bar,
 meta: { requiresLogin: true }
 }
]
 }
]
})
```

对应vue文件内

```
export default {
 mounted() {
 console.log(this.$route.meta);
 },
};
```

## VueRouter\_过渡动效-滚动行为

### 过渡动效

<router-view> 是基本的动态组件，所以我们可以用 <transition> 组件给它添加一些过渡效果。

```
<transition>
 <router-view></router-view>
</transition>
```

举例

html

```
<transition>
 <router-view></router-view>
</transition>
```

css

```
.v-enter {
 transform: translateX(1000px);
}
.v-enter-to {
 transform: translateX(0);
}
.v-enter-active {
 transition: all 0.5s;
}
```

### 滚动行为

使用前端路由，当切换到新路由时，想要页面滚到顶部，或者是保持原先的滚动位置，就像重新加载页面那样。vue-router 可以自定义路由切换时页面如何滚动。

注意：这个功能只在支持 `history.pushState` 的浏览器中可用。

当创建一个 Router 实例，你可以提供一个 `scrollBehavior` 方法：

```
const router = new VueRouter({
 routes,
 mode: 'history',
 scrollBehavior(to, from, savedPosition) {
 // return 期望滚动到哪个的位置
 return {
 x: 0,
 y: 0
 }
 }
})
```

`scrollBehavior` 方法接收 `to` 和 `from` 路由对象。第三个参数 `savedPosition` 当且仅当 `popstate` 导航（通过浏览器的 前进/后退 按钮触发）时才可用。

`scrollBehavior` 返回滚动位置的对象信息，长这样：

- { x: number, y: number }
- { selector: string, offset? : { x: number, y: number } } (offset 只在 2.6.0+ 支持)

```

scrollBehavior (to, from, savedPosition) {
 return { x: 0, y: 0 }
}

scrollBehavior (to, from, savedPosition) {
 if (to.hash) {
 return {
 selector: to.hash // selector 的 值为 hash值
 }
 }
}

```

举例：

```

const router = new VueRouter({
 routes,
 mode: 'history',
 scrollBehavior(to, from, savedPosition) {
 console.log(to.hash);
 // return 期望滚动到哪个的位置
 if (to.hash) {
 return {
 selector: to.hash // selector 的 值为 hash值
 }
 }
 }
})

```

点击跳转链接处

```
<router-link to="/demo2#a">此处可跳到demo2的a</router-link>
```

跳转到的地方

```
<div id="a">a可由首页跳到此处</div>
```

## Vuex\_State

Vuex是vue的状态管理工具，为了更方便的实现多个组件共享状态。

## 安装

```
npm install vuex --save
```

## 使用

```

import Vue from 'vue';
//引入vuex
import Vuex from 'vuex';

//使用Vuex
Vue.use(Vuex);
//根实例中注册store选项
const store = new Vuex.Store({
 //state为分享数据的置物架
 state: {
 //分享数据
 count: 0
 }
})

new Vue({
 store,
})

```

## State

单一状态树，使用一个对象就包含了全部的应用层级状态。

### 在Vue组件中获得Vuex状态

Vuex 通过store 选项，提供了一种机制将状态从跟组件“注入”到每一个子组件中（调用Vue.use(Vuex)）。

通过在根实例中注册store选项，该store实例会注入到根组件下的所有子组件中，且子组件能通过this.\$store访问。

```

<div class="home">
 {{ $store.state.count }}
</div>

```

### mapState 辅助函数

当一个组件需要获取多个状态时，将这些状态都声明为计算属性会有些重复和冗余。为了解决这个问题，我们可以使用mapState辅助函数帮助我们生成计算属性：

```

import { mapState } from 'vuex';
//可写为
computed: mapState(["count"]),
//也可写为
computed: {
 ...mapState(['count']),
},

```

要使用不同的名字：

```

computed: {
 ...mapState({
 //如果用分享值原名则可以这样写
 //count,
 //自己给分享值起的名:函数 state => {return state.count}
 storeCount: state => state.count,
 // 简写
 storeCount: 'count', //语法糖， 等同于 state => state.count
 }),
}

```

举例：vue组件

```

//获取mapState，以便从中找到count值
import { mapState } from "vuex";
export default {
 mounted() {
 console.log(mapState(["count"]));
 console.log(this.$store.state.count);
 },
 //分享名和自己起的名相同，可这样写
 //computed: mapState(["count"]),
 //也可以这样写
 // computed: {
 // ...mapState(["count"]),
 // },
 //如果分享名和自己起的名不相同
 computed: {
 ...mapState({
 //自己起的名：函数
 //storeCount: (state) => state.count,
 //语法糖 自己起的名字："分享名"
 storeCount: "count",
 }),
 },
};

```

## Vuex\_Getter

store的计算属性。getter的返回值会根据它的依赖被缓存起来，且只有当它的依赖值发生了改变才会被重新计算。

Getter 接收state作为其第一个参数、getters作为其第二个参数。

```

getters: {
 doubleCount (state) {
 return state.count * 2;
 }
}

```

## 通过属性访问

Getter会暴露为store.getters对象: this.\$store.getters.doubleCount

## 通过方法访问

也可以让getter返回一个函数，来实现给getter传参

```

getters: {
 addCount: state => num => state.count + num;
}
//相当于
getters:{
 addCount: (state) => {
 return num => state.count + num;
 }
}

```

使用getters

```
this.$store.addCount(3);
```

或者

```
export default {
 computed: {
 ...mapGetters({
 addCount: "addCount",
 }),
 },
};
```

举例：

```
<div>
 <button @click="$store.state.count++">+1</button>
 {{ storeCount }}
 {{ addCount(3) }}
</div>
```

使用的vue文件

```
import { mapState, mapGetters } from "vuex";
export default {
 computed: {
 ...mapState({
 //自己起的名: 函数
 //storeCount: (state) => state.count,
 //语法糖 自己起的名字: "分享名"
 storeCount: "count",
 }),
 ...mapGetters({
 addCount: "addCount",
 }),
 },
};
```

store.js

```
export default new Vuex.Store({
 state: {
 count: 0,
 studentList: []
 },
 getters: {
 addCount: (state) => {
 return num => state.count + num;
 }
 }
});
```

## mapGetters 辅助函数

```
import { mapGetters } from 'vuex';

export default {
 computed: {
 ...mapGetters([
 'doubleCount',
 'addCount',
])
 }
}
```

如果你想将一个 getter 属性另取一个名字，使用对象形式：

```
mapGetters({
 // 把 `this.doneCount` 映射为 `this.$store.getters.doneTodosCount`
 storeDoubleCount: 'doubleCount'
})
```

# Vuex\_Mutation

在Vuex严格模式中，更改 Vuex 的 store 中的状态的唯一方法是提交 mutation。

```
const store = new Vuex.Store({
 state: {
 count: 1
 },
 mutations: {
 increment (state) {
 // 变更状态
 state.count++
 }
 }
})
```

不能直接调用一个mutation handler。这个选项更像是事件注册：“当触发一个类型为 increment 的mutation时，调用此函数。”：

```
this.$store.commit('increment');
```

## 在组件中提交 Mutation

除了在组件中使用 `this.$store.commit('xxx')` 提交 mutation之外，还可以使用 `mapMutations` 辅助函数：

```
import { mapMutations } from 'vuex'

export default {
 // ...
 methods: {
 ...mapMutations([
 'increment', // 将 `this.increment()` 映射为 `this.$store.commit('increment')`
]),
 ...mapMutations({
 add: 'increment' // 将 `this.add()` 映射为 `this.$store.commit('increment')`
 })
 }
}
```

举例：

store.js

```
export default new Vuex.Store({
 state: {
 count: 0,
 studentList: []
 },
 mutations: {
 countPlus(state) {
 state.count++;
 },
 countPlusTwo(state) {
 state.count += 2;
 }
 }
});
```

组件中

```
//获取mapMutations
import { mapState, mapGetters, mapMutations } from "vuex";
export default {
 methods: {
 //多个方法可以在数组里加入，也可以单独写一个mapMutations放入方法名
 ...mapMutations(["countPlus", "countPlusTwo"]),
 // ...mapMutations(["countPlusTwo"]),
 //也可单独写法方法，放入vuex定义好的Mutation名
 plus() {
 this.$store.commit("countPlus");
 },
 },
};
```

## 提交载荷 (Payload)

你可以向`store.commit`传入额外的参数，即mutation的载荷 (payload)：

```
mutations: {
 increment (state, n) {
 state.count += n
 }
}
```

组件中使用

```
methods: {
 store.commit('increment', 10)
}
```

在大多数情况下，载荷应该是一个对象，这样可以包含多个字段并且记录的mutation会更易读：

```
mutations: {
 increment (state, payload) {
 state.count += payload.amount
 }
}
```

```
store.commit('increment', {
 amount: 10
})
```

举例：

`store.js`

```
export default new Vuex.Store({
 state: {
 count: 0,
 studentList: []
 },
 mutations: {
 countPlusN(state, n) {
 state.count += n;
 },
 countPlusPayload(state, payload) {
 state.count += payload.n;
 }
 }
});
```

组件中

```
//获取mapMutations
import {mapMutations} from "vuex";
export default {
 methods: {
 //Mutation中方法加参数
 plusTen() {
 //第①种方法
 //this.$store.commit("countPlusN", 10);
 //第②种方法
 this.$store.commit("countPlusPayload", { n: 10 });
 },
 },
};
```

## 对象风格的提交方式

提交 mutation 的另一种方式是直接使用包含 type 属性的对象：

vue组件中

```
store.commit({
 type: 'increment',
 amount: 10
})
```

当使用对象风格的提交方式，整个对象都作为载荷传给 mutation 函数，因此 handler 保持不变：

```
mutations: {
 increment (state, payload) {
 state.count += payload.amount
 }
}
```

## 使用常量替代 Mutation 事件类型

把这些常量放在单独的文件中可以让你的代码合作者对整个 app 包含的 mutation 一目了然：

```
// mutation-types.js
//提交的mutation 名字都是大写
export const COUNT_INCREMENT = 'COUNT_INCREMENT'

// store.js
import Vuex from 'vuex'
import { COUNT_INCREMENT } from './mutation-types'

const store = new Vuex.Store({
 state: { ... },
 mutations: {
 //由于是使用的导出常量作为方法名，所以要加中括号
 [COUNT_INCREMENT] (state) {
 // ...
 }
 }
})
```

举例：

mutation-types.js

```
export const COUNT_PLUS = 'COUNT_PLUS';
```

store.js

```

import Vue from 'vue';
import Vuex from 'vuex';
import {
 COUNT_PLUS
} from './store/mutation-types';

//vue使用Vuex
Vue.use(Vuex);

//注册Store选项，并导出
export default new Vuex.Store({
 state: {
 count: 0,
 studentList: []
 },
 mutations: {
 [COUNT_PLUS](state) {
 state.count++;
 }
 }
});

```

组件中使用

```

//获取mapMutations
import { mapState, mapGetters, mapMutations } from "vuex";
import { COUNT_PLUS } from "../store/mutation-types";
export default {
 mounted() {},
 methods: {
 plus() {
 //this.$store.commit("countPlus");
 this.$store.commit(COUNT_PLUS);
 },
 },
};

```

用不用常量取决于自己，在需要多人协作的大型项目中，这会很有帮助。

## Mutation 需遵守 Vue 的响应规则

既然 Vuex 的 store 中的状态是响应式的，那么当我们变更状态时，监视状态的 Vue 组件也会自动更新。这也意味着 Vuex 中的 mutation 也需要与使用 Vue 一样遵守一些注意事项：

- 最好提前在你的 store 中初始化好所有所需属性。
- 当需要在对象上添加新属性时，你应该
  - 使用 `Vue.set(obj, 'newProp', 123)`，或者
  - 以新对象替换老对象。例如，利用对象展开运算符我们可以这样写：

```
state.obj = { ...state.obj, newProp: 123 }
```

举例：

`store.js`

```

export default new Vuex.Store({
 state: {
 obj: {
 a: 1
 }
 },
});

```

组件中使用

```

export default {
 mounted() {},
 computed: {
 ...mapState({
 //用于验证
 obj: "obj",
 }),
 },
 methods: {
 changeObj() {
 this.$store.state.obj = { ...this.$store.state.obj, b: 123 };
 },
 },
};

```

## 表单处理

在Vuex的state上使用v-model时，由于会直接更改state的值，所以Vue会抛出错误。

如果想要使用双向数据的功能，就需要自己模拟一个v-model: :value="msg" @input="updateMsg"。

## 双向绑定的计算属性

上面的做法，比v-model本身繁琐很多，所以我们还可以使用计算属性的setter来实现双向绑定：

```

<input v-model="msg">

computed: {
 msg: {
 get () {
 return this.$store.state.obj.msg;
 },
 set (value) {
 this.$store.commit(UPDATE_MSG, { value });
 }
 }
}

```

举例：

store.js

```

export default new Vuex.Store({
 state: {
 msg: ''
 },
 mutations: {
 input(state, value) {
 state.msg = value;
 }
 }
});

```

组件中使用

```

<template>
 <div>
 <!-- <hr />
 <input type="text" :value="msg" @input="input" />
 {{ msg }} -->
 <hr />
 <input type="text" v-model="msg" />
 {{ msg }}
 </div>
</template>

```

```
//获取mapMutations
import { mapState, mapGetters, mapMutations } from "vuex";
import { COUNT_PLUS } from "../store/mutation-types";
export default {
 computed: {
 ...mapState({
 //msg: "msg",
 }),
 //第二种方法 实现v-model
 msg: {
 get() {
 return this.$store.state.msg;
 },
 set(value) {
 this.$store.commit("input", value);
 },
 },
 },
 methods: {
 //第一种方法 实现v-model
 input(e) {
 this.$store.commit("input", e.target.value);
 },
 },
};
```

## Mutation 必须是同步函数

要记住 `mutation` 必须是同步函数。why?

```
mutations: {
 [COUNT_INCREMENT] (state) {
 setTimeout(() => {
 state.count++;
 }, 1000)
 },
}
```

执行上端代码，我们会发现更改`state`的操作是在回调函数中执行的，这样会让我们的代码在`devtools`中变的不好调试：当 `mutation` 触发的时候，回调函数还没有被调用，`devtools` 不知道什么时候回调函数实际上被调用，任何在回调函数中进行的状态的改变都是不可追踪的。

## 严格模式

开启严格模式，仅需在创建 `store` 的时候传入 `strict: true`:

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

在严格模式下，无论何时发生了状态变更且不是由 `mutation` 函数引起的，将会抛出错误。这能保证所有的状态变更都能被调试工具跟踪到。

## 开发环境与发布环境

不要在发布环境下启用严格模式！严格模式会深度监测状态树来检测不合规的状态变更，要确保在发布环境下关闭严格模式，以避免性能损失。

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

## Vuex\_Action

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

Action 函数接受一个与 store 实例具有相同方法和属性的 context 对象，因此你可以调用 context.commit 提交一个 mutation，或者通过 context.state 和 context.getters 来获取 state 和 getters：

```
const store = new Vuex.Store({
 state: {
 count: 0
 },
 mutations: {
 increment (state) {
 state.count++
 }
 },
 actions: {
 increment (context) {
 context.commit('increment')
 }
 }
})
```

## 分发Action

```
store.dispatch('increment')
```

虽然和mutation差不多，但是在action中，可以执行异步操作，但是mutation中不行！！！因为mutation store.js

```
actions: {
 incrementAsync ({ commit }) {
 setTimeout(() => {
 commit('increment')
 }, 1000)
 }
}
```

举例：

store.js

```
export default new Vuex.Store({
 //要是生产模式的话就关闭严格模式
 strict: process.env.NODE_ENV !== 'production',
 state: {
 count: 0,
 },
 mutations: {
 countPlusPayload(state, payload) {
 state.count += payload.n;
 }
 },
 actions: {
 countPlusNAction(context, payload) {
 setTimeout(() => {
 context.commit('countPlusPayload', payload);
 }, 1000)
 }
 }
});
```

组件中使用

```

import { mapState, mapActions } from "vuex";
export default {
 computed: {
 ...mapState({
 //语法糖 自己起的名字: "分享名"
 storeCount: "count",
 }),
 },
 methods: {
 ...mapActions(["countPlusNAction"]),
 //Mutation中方法加参数
 plusTen() {
 //第①种Action提交方法
 //this.$store.dispatch("countPlusNAction", { n: 10 });
 //第②种Action提交方法
 this.countPlusNAction({ n: 10 });
 }
 },
};

```

## 组合 Action

Action 通常是异步的，那么如何知道 action 什么时候结束呢？

store.js 中

```

actions: {
 actionA ({ commit }) {
 return new Promise((resolve, reject) => {
 setTimeout(() => {
 commit('someMutation')
 resolve()
 }, 1000)
 })
 }
}

```

组件中

```

store.dispatch('actionA').then(() => {
 // ...
})

```

举例：

store.js

```

export default new Vuex.Store({
 strict: process.env.NODE_ENV !== 'production',
 state: {
 count: 0,
 },
 mutations: {
 countPlusPayload(state, payload) {
 state.count += payload.n;
 },
 },
 actions: {
 countPlusNAction(context, payload) {
 return new Promise((resolve, reject) => {
 setTimeout(() => {
 context.commit('countPlusPayload', payload);
 //promise返回值
 resolve();
 }, 1000)
 })
 }
 }
});

```

组件中使用

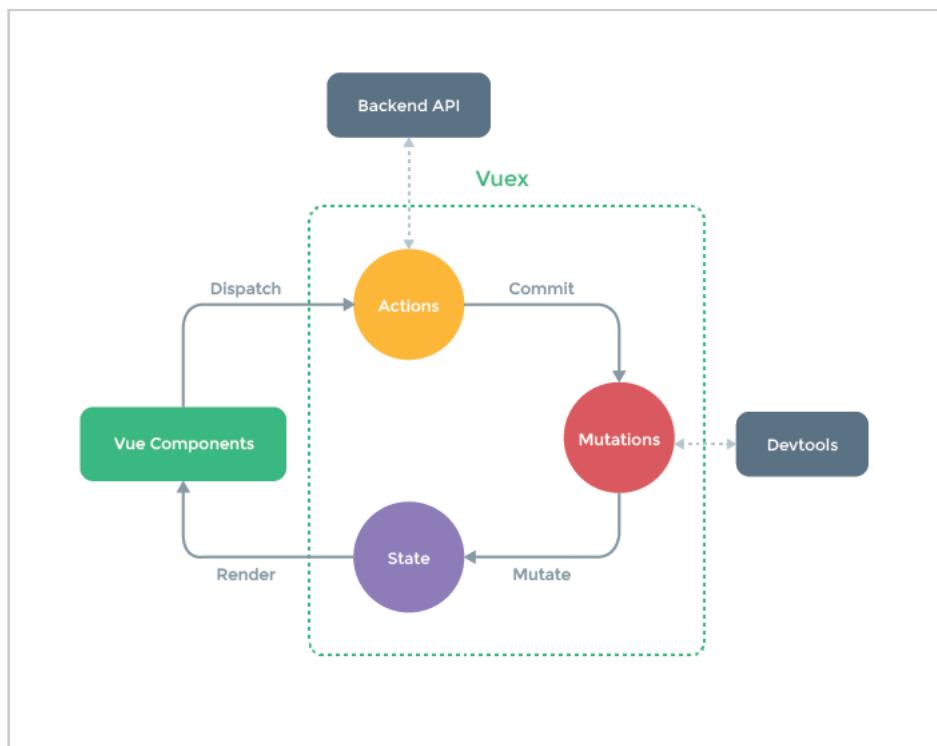
```

import { mapState, mapActions } from "vuex";
export default {
 computed: {
 ...mapState({
 //语法糖 自己起的名字: "分享名"
 storeCount: "count",
 }),
 },
 methods: {
 plusTen() {
 this.$store.dispatch("countPlusNAction", { n: 10 }).then(() => {
 alert("ok");
 });
 },
 },
};

```

## Vuex 管理模式

如果state改变不需要异步，则Vue Components可以直接指向Mutations



## Vuex\_Module(模块)

由于使用单一状态树，应用的所有状态会集中到一个比较大的对象。当应用变得非常复杂时，`store` 对象就有可能变得相当臃肿。

为了解决以上问题，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`。

```

modules: {
 a,
 b
}

```

- 获取 state: `this.$store.state.moduleName.xxx`
- 获取 getter: `this.$store.getters.xxx`
- 提交 mutation: `this.$store.commit('xxx');`
- 分发 action: `this.$store.dispatch('xxx');`
- 可以通过`mapXXX`的方式拿到`getters`、`mutations`、`actions`，但是不能拿到`state`，如果想通过这种方式获得`state`，需要加命名空间。

# 命名空间

在命名空间内添加 `namespaced: true` 的方式使其成为带命名空间的模块。

- 获取 state: `this.$store.state.moduleName.xxx`
- 获取 getter: `this.$store['moduleName/getters'].xxx`
- 提交 mutation: `this.$store.commit('moduleName/xxx');`
- 分发 action: `this.$store.dispatch('moduleName/xxx');`
- 可以通过`mapXXX`的方式获取到state、getters、mutations、actions。

举例:

各命名组件count.js:

```
export default {
 namespaced: true,
 state: {
 count: 0,
 },
 getters: {
 ...
 },
 mutations: {
 ...
 },
 actions: {
 ...
 }
};
```

组件使用

```
import { mapState, mapGetters, mapMutations, mapActions } from "vuex";
import { COUNT_PLUS } from "../store/mutation-types";
export default {
 mounted() {},
 computed: {
 ...mapState({
 //语法糖 自己起的名字: "分享名"
 storeCount: "count",
 //用于验证
 obj: "obj",
 //msg: "msg",
 }),
 //count为命名空间
 ...mapGetters("count", {
 addCount: "addCount",
 }),
 },
 methods: {
 //命名空间写法 空间名 count
 ...mapMutations("count", ["countPlus", "countPlusTwo"]),
 plus() {
 //使用命名空间 count为命名空间
 this.$store.commit("count/" + COUNT_PLUS);
 },
 //Mutation中方法加参数
 plusTen() {
 //使用命名空间 count为命名空间
 this.$store.dispatch("count/countPlusNAction", { n: 10 }).then(() => {
 alert("ok");
 });
 },
 input(e) {
 this.$store.commit("input", e.target.value);
 },
 },
};
```

# 模块的局部状态

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。

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。

对于模块内部的 getter，根节点状态会作为第三个参数暴露出来。

举例：模块 count.js 中

```
export default {
 namespaced: true,
 state: {
 count: 0,
 },
 getters: {
 countDouble: (state, getters, rootState) => {
 // 可以打印模块状态
 console.log(getters, rootState);
 return state.count * 2
 },
 countAdd: state => num => state.count + num,
 },
}
```