

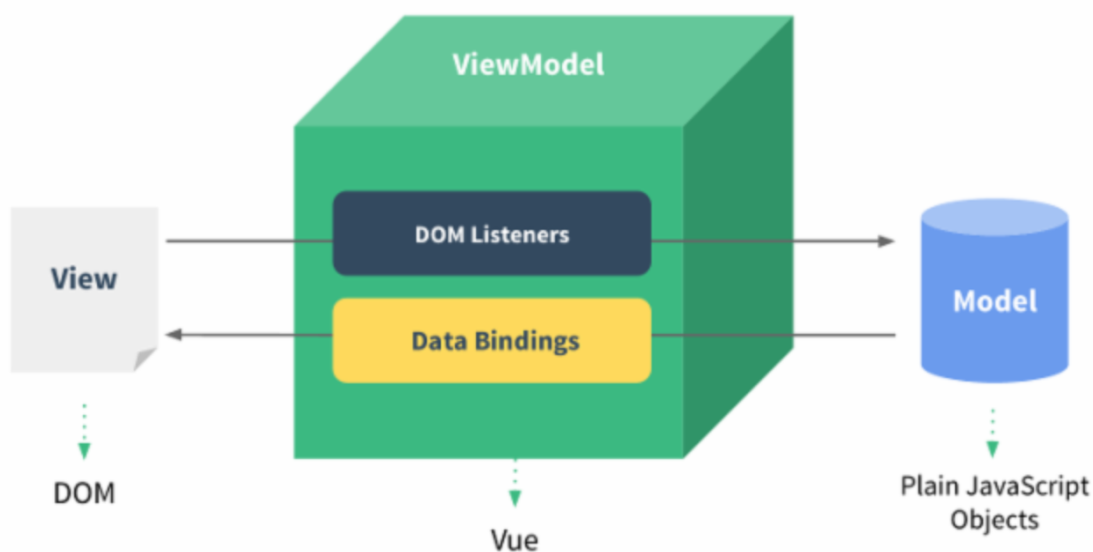
Vue.js

1. Vue.js的介绍

Vue.js是一套用于构建用户界面的**渐进式框架**，是一套轻量级MVVM框架

MVVM (Model-View-ViewModel) 模式将数据双向绑定 (data-binding) 作为核心思想，View 和 Model 之间没有联系，它们通过 ViewModel 这个桥梁进行交互。

【view】：视图层（UI 用户界面）
【viewModel】：业务逻辑层（一切 js 可视为业务逻辑）
【Model】：数据层（存储数据及对数据的处理如增删改查）



2. Vue.js的优点

1. 简单轻巧，功能强大，拥有非常容易上手的 API；
2. 可组件化 和 响应式设计；
3. 实现数据与结构分离，高性能，易于浏览器的加载速度；

基于虚拟dom，一种可以预先通过JavaScript进行各种计算，把最终的DOM操作计算出来并优化的技术，由于这个DOM操作属于预处理操作，并没有真实的操作DOM，所以叫做虚拟DOM

4. MVVM 模式，数据双向绑定，减少了 DOM 操作，将更多精力放在数据和业务逻辑上。

3. Vue.js的环境搭建

前期学习使用vue.js的环境搭建主要含有一下几个步骤：

- 安装HbuilderX
- 创建Vue项目
- 安装vue-router
- 配置router

1. 安装HbuilderX

通过官网<https://www.dcloud.io/hbuilderx.html>，下载相应版本的HbuilderX，解压下载的压缩包即可完成安装。

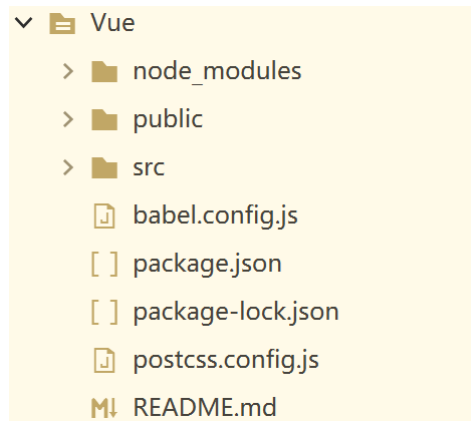
2. 创建Vue项目

使用HbuilderX创建一个vue-cli默认项目，操作步骤如下：

- 打开HbuilderX，点击“文件 ---> 新建 ---> 项目”
- 在选择模板中选择“vue项目(2.6.10)”




创建完成后的目录结构如下所示：



3. 安装vue-router

HbuilderX创建的vue-cli模板文件中没有vue-router,需要我们自己添加。

选中当前项目，打开HbuilderX控制台（界面左下角的  图标，或者Alt+C快捷键），输入命令回车安装即可。

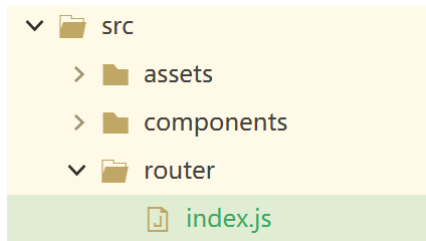
```
npm install vue-router
```

安装完成后，可以在package.json文件中看到安装成功的依赖。

```
"dependencies": {  
  "core-js": "^2.6.5",  
  "vue": "^2.6.10",  
  "vue-router": "^3.5.1"  
},
```

4. 配置router

1. 在项目的src目录下创建一个router目录，在router目录下创建一个index.js文件



2. router目录下的index.js文件模板

```
// 0. 如果使用模块化机制编程，导入Vue和VueRouter，要调用 Vue.use(VueRouter)  
import Vue from 'vue'  
import VueRouter from 'vue-router'  
// 1. 定义（路由）组件。  
// 可以从其他文件 import 进来  
  
Vue.use(VueRouter)  
  
// 2. 定义路由  
// 每个路由应该映射一个组件。  
const routes = [];  
  
// 3. 创建 router 实例，然后传 `routes` 配置  
const router = new VueRouter({  
  routes  
})  
  
export default router
```

3. src目录下的main.js文件模板

```
import Vue from 'vue'  
import App from './App.vue'  
import router from './router'  
  
Vue.config.productionTip = false //消息提示的环境配置，设置为开发环境或者生产环境，当前为开发模式  
  
// 4. 创建和挂载根实例。  
new Vue({  
  router, //通过 router 配置参数 注入路由， 从而让整个应用都有路由功能  
  render: h => h(App),  
}).$mount('#app')
```

4. 创建Vue的实例

Vue.js 的核心是一个允许采用简洁的模板语法来声明式地将数据渲染进 DOM 的系统：

```
<template>
  <div id="app">
    <h1>{{msg}}</h1>
  </div>
</template>
```

```
// 导出当前vue实例
export default {
  name: 'app',
  data(){
    return {
      msg: 'hello'
    }
  }
}
```

5. Vue.js的属性与方法

当一个 Vue 实例被创建时，它将 `data` 对象中的所有的 property 加入到 Vue 的**响应式系统**中。当这些 property 的值发生改变时，视图将会产生“响应”，即匹配更新为新的值

```
// 我们的数据对象
var data = { a: 1 }

// 该对象被加入到一个 vue 实例中
export default {
  name: 'app',
  data(){
    return data
  }
}

// 获得这个实例上的 property
// 返回源数据中对应的字段
// 在export default 中使用this，在这里的含义即该vue实例
this.a == data.a // => true

// 设置 property 也会影响到原始数据
this.a = 2
data.a // => 2

// .....反之亦然
data.a = 3
this.a // => 3
```

注意：

1. 只有当实例被创建时就已经存在于data 中的属性才是**响应式**的。也就是说如果你在创建实例后添加一个新的属性，比如：

```
<template>
  <div id="app">
    <h1>{{b}}</h1>
  </div>
</template>
```

```
export default {
  name: 'app',
  data(){
    return {} // 未声明任何属性（变量）
  },
  created(){ // vue实例被创建后，调用的钩子函数
    // 在实例被创建之后声明使用b
    // b 的改动将不会触发任何视图的更新
    this.b = 'hi';
    setTimeout(()=>{
      this.b = '12';
    },2000)
  }
}
```

注意:

1. 如果存在实例创建前声明的变量，比如a。当a在b改变之后，发生改变，视图重新渲染，会将b的值进行更新。
2. 如果你知道你会在晚些时候需要一个 property，但是一开始它为空或不存在，那么你仅需要设置一些初始值

```
data(){
  return {
    newTodoText: '',
    visitCount: 0,
    hideCompletedTodos: false,
    todos: [],
    error: null
  }
}
```

2. 除了数据属性，Vue实例还暴露了一些有用的实例属性与方法。它们都有前缀 \$，以便与用户定义的属性区分开来

```
var data = { a: 1 }
// 该对象被加入到一个 vue 实例中
export default {
  name: 'app',
  data(){
    return data
  },
  created(){ // vue实例被创建后，调用的钩子函数
    this.$data === data // => true

    // $watch 是一个实例方法
    this.$watch('a', function (newValue, oldValue) {
      // 这个回调将在 `vm.a` 改变后调用
    })
  }
}
```

```
    }  
  }  
}
```

1. 实例属性

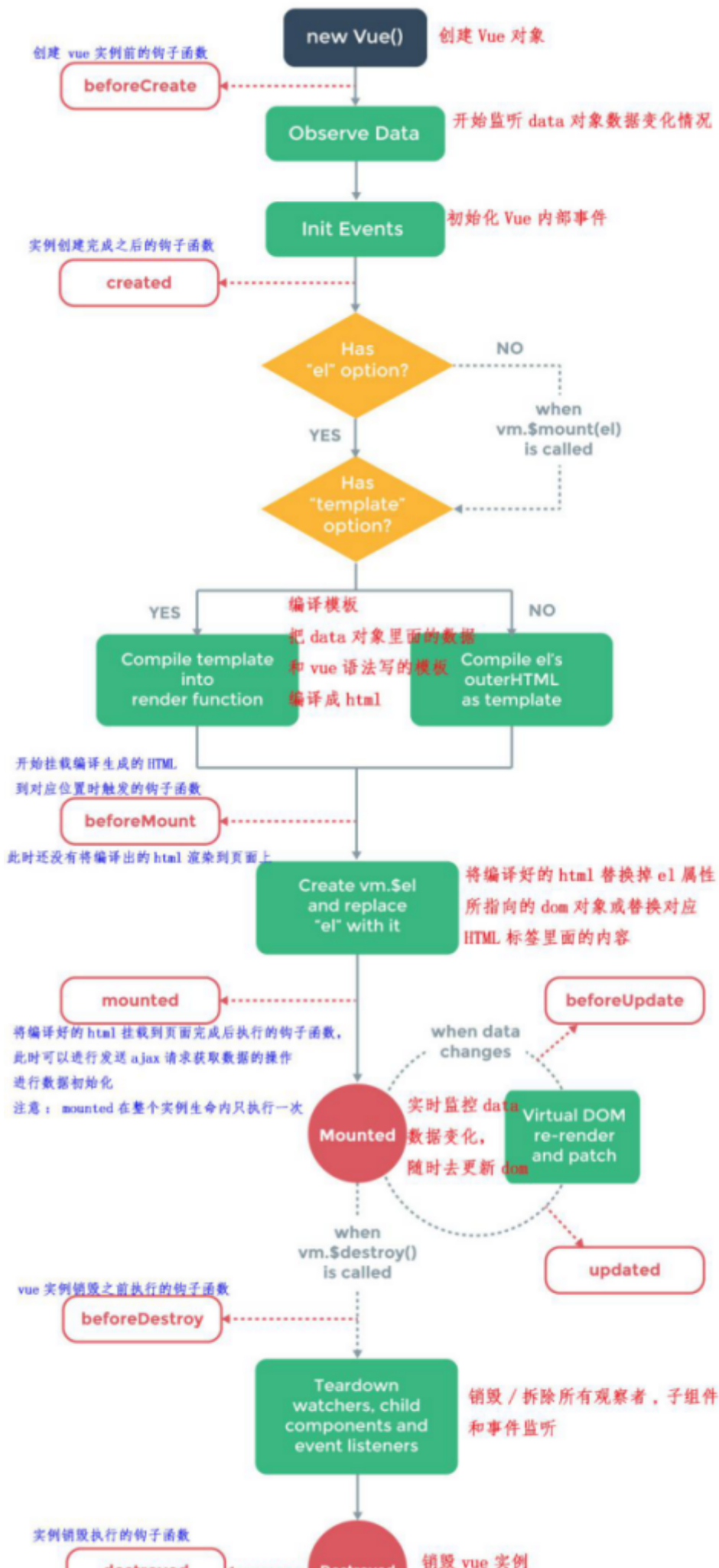
实例属性	类型	详细
vm.\$data	Object	Vue 实例观察的数据对象。Vue 实例代理了对其 data 对象属性的访问。
vm.\$props	Object	当前组件接收到的 props 对象。Vue 实例代理了对其 props 对象属性的访问。
vm.\$el	HTMLElement	Vue 实例使用的根 DOM 元素。
vm.\$options	Object	用于当前 Vue 实例的初始化选项。
vm.\$parent	Vue instance	父实例，如果当前实例有的话。
vm.\$root	Vue instance	当前组件树的根 Vue 实例。如果当前实例没有父实例，此实例将会是其自己。
vm.\$children	Array<Vue instance>	当前实例的直接子组件。需要注意 \$children 并不保证顺序，也不是响应式的。
vm.\$slots	{ [name: string]: ?Array<VNode> }	用来访问被插槽分发的内容。每个具名插槽 有其相应的属性 (例如：v-slot:foo 中的内容将会在 vm.\$slots.foo 中被找到)。
vm.\$scopedSlots	{ [name: string]: props => Array<VNode> undefined }	用来访问作用域插槽。对于包括 默认 slot 在内的每一个插槽，该对象都包含一个返回相应 VNode 的函数。
vm.\$refs	Object	一个对象，持有注册过 ref 特性的 所有 DOM 元素和组件实例。
vm.\$isServer	boolean	当前 Vue 实例是否运行于服务器。
vm.\$attrs	{ [key: string]: string }	包含了父作用域中不作为 prop 被识别 (且获取) 的特性绑定 (class 和 style 除外)。当一个组件没有声明任何 prop 时，这里会包含所有父作用域的绑定 (class 和 style 除外)，并且可以通过 v-bind="\$attrs" 传入内部组件——在创建高级别的组件时非常有用。
vm.\$listeners	{ [key: string]: Function Array<Function> }	包含了父作用域中的 (不含 .native 修饰器的) v-on 事件监听器。它可以通过 v-on="\$listeners" 传入内部组件——在创建更高层次的组件时非常有用。

2. 实例方法

实例方法	用法
vm.\$watch(expOrFn, callback, [options])	观察 Vue 实例变化的一个表达式或计算属性函数。回调函数得到的参数为新值和旧值。表达式只接受监督的键路径。对于更复杂的表达式，用一个函数取代。
vm.\$set(target, propertyName/index, value)	这是全局 Vue.set 的别名。
vm.\$delete(target, propertyName/index)	这是全局 Vue.delete 的别名。
vm.\$on(event, callback)	监听当前实例上的自定义事件。事件可以由 vm.\$emit 触发。回调函数会接收所有传入事件触发函数的额外参数。
vm.\$once(event, callback)	监听一个自定义事件，但是只触发一次，在第一次触发之后移除监听器。
vm.\$off([event, callback])	移除自定义事件监听器。
vm.\$emit(eventName, [...args])	触发当前实例上的事件。附加参数都会传给监听器回调。
vm.\$mount(elementOrSelector)	如果 Vue 实例在实例化时没有收到 el 选项，则它处于“未挂载”状态，没有关联的 DOM 元素。可以使用 vm.\$mount() 手动地挂载一个未挂载的实例。
vm.\$forceUpdate()	迫使 Vue 实例重新渲染。注意它仅仅影响实例本身和插入插槽内容的子组件，而不是所有子组件。
vm.\$nextTick([callback])	将回调延迟到下次 DOM 更新循环之后执行。在修改数据之后立即使用它，然后等待 DOM 更新。它跟全局方法 Vue.nextTick 一样，不同的是回调的 this 自动绑定到调用它的实例上。
vm.\$destroy()	完全销毁一个实例。清理它与其它实例的连接，解绑它的全部指令及事件监听器。 触发 beforeDestroy 和 destroyed 的钩子。

6. Vue.js的生命周期

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





生命周期	详细
beforeCreate	在实例初始化之后，数据观测 (data observer) 和 event/watcher 事件配置之前被调用。
created	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。然而，挂载阶段还没开始，\$el 属性目前不可见。
beforeMount	在挂载开始之前被调用：相关的 render 函数首次被调用。该钩子在服务器端渲染期间不被调用。
mounted	el 被新创建的 vm.\$el 替换，并挂载到实例上去之后调用该钩子。如果 root 实例挂载了一个文档内元素，当 mounted 被调用时 vm.\$el 也在文档内。
beforeUpdate	数据更新时调用，发生在虚拟 DOM 打补丁之前。这里适合在更新之前访问现有的 DOM，比如手动移除已添加的事件监听器。
updated	由于数据更改导致的虚拟 DOM 重新渲染和打补丁，在这之后会调用该钩子。
activated	keep-alive 组件激活时调用。
deactivated	keep-alive 组件停用调用。
beforeDestroy	实例销毁之前调用。在这一步，实例仍然完全可用。
destroyed	Vue 实例销毁后调用。调用后，Vue 实例指示的所有东西都会解绑定，所有的事件监听器会被移除，所有的子实例也会被销毁。
errorCaptured	当捕获一个来自子孙组件的错误时被调用。

keep-alive组件是Vue的内置组件，能在组件切换过程中将状态保留在内存中，防止重复渲染DOM

比如：

1. 用于保持首页五个板块的消息列表状态，而不用在点击进入详情后回退列表时再次刷新
2. 用来保持用户的登陆状态

7. 模块语法

Vue.js 使用了基于 HTML 的模板语法，允许开发者声明式地将 DOM 绑定至底层 Vue 实例的数据。所有 Vue.js 的模板都是合法的 HTML，所以能被遵循规范的浏览器和 HTML 解析器解析。

在底层的实现上，Vue 将模板编译成虚拟 DOM 渲染函数。结合响应系统，Vue 能够智能地计算出最少需要重新渲染多少组件，并把 DOM 操作次数减到最少。

1. 插值

1. 文本

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

```
<span>Message: {{ msg }}</span>
<!-- Mustache 标签将会被替代为对应数据对象上msg属性的值。无论何时，绑定的数据对象上
sg属性发生了改变，插值处的内容都会更新。 -->

<span v-once>这个将不会改变: {{ msg }}</span>
<!-- 通过使用 v-once 指令，你也能执行一次性地插值，当数据改变时，插值处的内容不会更
新。-->
<!-- 但请留心这会影响该节点上的其它数据绑定-->
```

2. 原始HTML

双大括号会将数据解释为普通文本，而非 HTML 代码。为了输出真正的 HTML，你需要使用 v-html指令

```
<!-- rawHtml : '<span style="color:red">this is should be red</span>' -
->
<p>Using mustaches: {{ rawHtml }}</p>
<p>Using v-html directive: <span v-html="rawHtml"></span></p>
```


3. dom元素（标签）自带的属性

Mustache 语法（双大括号形式）不能作用在 HTML 标签元素自带的属性上，遇到这种情况应该使用v-bind指令

```
<div v-bind:id="dynamicId"></div>

<!-- 对于布尔标签自带属性(它们只要存在就意味着值为true)，v-bind效果略有不同-->
<!-- 如果isButtonDisabled的值是null、undefined或false，
      则disabled属性甚至不会被包含在渲染出来的 <button>元素中 -->
<button v-bind:disabled="isButtonDisabled">Button</button>
```

4. JavaScript表达式

对于所有的数据绑定，Vue.js 都提供了完全的 JavaScript 表达式支持。

```
{{ number + 1 }}

{{ ok ? 'YES' : 'NO' }}

{{ message.split('').reverse().join('') }}

<div v-bind:id="'list-' + id"></div>

<!-- 这些表达式会在所属 Vue 实例的数据作用域下作为 JavaScript 被解析。 -->
<!-- 有个限制就是，每个绑定都只能包含单个表达式，所以下面的例子都不会生效。-->

<!-- 这是语句，不是表达式 -->
{{ var a = 1 }}

<!-- 流控制也不会生效，请使用三元表达式 -->
{{ if (ok) { return message } }}
```

2. 指令

指令 (Directives) 是带有 `v-` 前缀的特殊 attribute。指令 attribute 的值预期是**单个 JavaScript 表达式** (v-for是例外情况)。指令的职责是，当表达式的值改变时，将其产生的连带影响，响应式地作用于 DOM。

```
<p v-if="seen">现在你看到我了</p>

<!-- <p v-if="seen">现在你看到我了</p> -->
```

1. 参数

一些指令能够接收一个“参数”，在指令名称之后以冒号表示。

```
<!-- 例一： v-bind指令可以用于响应式地更新 HTML 标签自带属性 -->
<!-- 在这里 href 是参数，告知 v-bind 指令将该元素的 href attribute 与表达式 url 的值绑定。-->
<a v-bind:href="url">...</a>

<!-- 例子二： v-on指令，它用于监听 DOM 事件： -->
<!-- 参数是监听的事件名 -->
<a v-on:click="doSomething">...</a>
```

2. 动态参数

从 2.6.0 开始，可以用方括号括起来的 JavaScript 表达式作为一个指令的参数：

```
<!-- 注意，参数表达式的写法存在一些约束 -->
<!-- 这里的 attributeName 会被作为一个 JavaScript 表达式进行动态求值，求得的值将会作为最终的参数来使用。例如，如果你的 vue 实例有一个 data property attributeName，其值为 "href"，那么这个绑定将等价于 v-bind:href。-->
<a v-bind:[attributeName]="url"> ... </a>

<!-- 使用动态参数为一个动态的事件名绑定处理函数 -->
<!-- 在这个示例中，当 eventName 的值为 "focus" 时，v-on:[eventName] 将等价于 v-on:focus。-->
<a v-on:[eventName]="doSomething"> ... </a>
```

3. 缩写

`v-` 前缀作为一种视觉提示，用来识别模板中 Vue 特定的 attribute。当你在使用 Vue.js 为现有标签添加动态行为 (dynamic behavior) 时，`v-` 前缀很有帮助，然而，对于一些频繁用到的指令来说，就会感到使用繁琐。同时，在构建由 Vue 管理所有模板的单页面应用程序 (SPA - single page application) 时，`v-` 前缀也变得没那么重要了。因此，Vue 为 `v-bind` 和 `v-on` 这两个最常用的指令，提供了特定简写：

1. v-bind缩写

```
<!-- 完整语法 -->
<a v-bind:href="url">...</a>

<!-- 缩写 -->
<a :href="url">...</a>

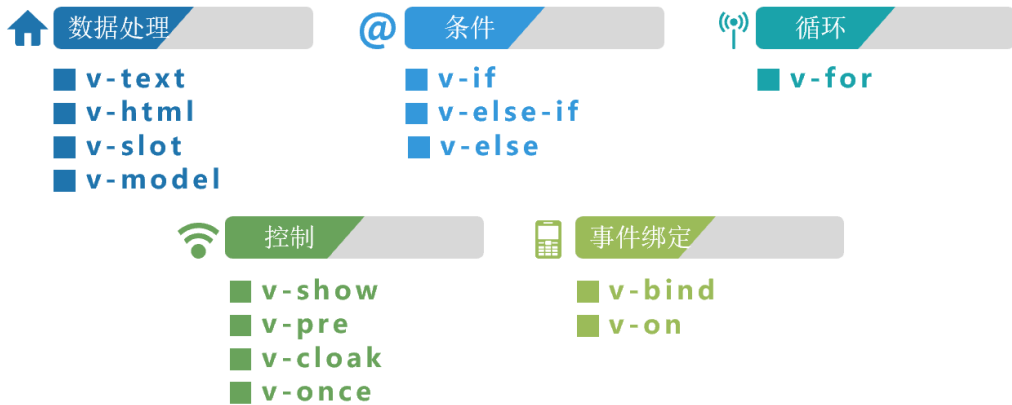
<!-- 动态参数的缩写 (2.6.0+) -->
<a :[key]="url"> ... </a>
```

2. v-on缩写

```
<!-- 完整语法 -->
<a v-on:click="doSomething">...</a>

<!-- 缩写 -->
<a @click="doSomething">...</a>

<!-- 动态参数的缩写 (2.6.0+) -->
<a @[event]="doSomething"> ... </a>
```



8. 样式绑定

操作元素的 class 列表和内联样式是数据绑定的一个常见需求。因为它们都是标签自带属性，所以我们可以用 `v-bind` 处理它们：只需要通过表达式计算出字符串结果即可。不过，字符串拼接麻烦且易错。因此，在将 `v-bind` 用于 `class` 和 `style` 时，Vue.js 做了专门的增强。表达式结果的类型除了字符串之外，还可以是**对象或数组**。

1. 绑定 HTML Class

1. 对象语法

可以传给 `v-bind:class` 一个对象，以动态地切换 class：

```
<template>
  <div class="hello">
    <div v-bind:class="{ active: isActive }">背景色</div>
    <div v-bind:class="{ active: isActive,div_circle: isCircle }">背景
    色/圆</div>
    <div class = "fontSize20"
      v-bind:class="{ active: isActive,div_circle: isCircle }">背景
    色/圆</div>
    <div v-bind:class="styleObj">对象传值</div>
  </div>
</template>

<script>
export default {
  name: 'HelloWorld',
  props: {
    msg: String
  },
  data() {
    return {
      isActive: true,
      isCircle: true,
      active: 'active',
      styleObj: { active: false, div_circle: true },
    }
  },
}
</script>

<!-- 添加 "scoped" 属性，可以限制CSS只在该组件起效果 -->
<style scoped>
  .active{
```

```

        background-color: #adadda;
    }
    .div_circle{
        width: 200px;
        height: 200px;
        border-radius: 50%;
        border: 2px solid #D19A66;
        line-height: 200px;
        text-align: center;
    }
    .div_box{
        width: 200px;
        height: 200px;
        border: 2px dashed #D3869B;
        line-height: 200px;
        text-align: center;
    }
    .fontSize20{
        font-size: 20px;
    }
}
</style>

```

2. 数组语法

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

```

<template>
  <div class="hello">
    <div v-bind:class="[ active, 'div_box']">背景色/方形</div>
    <div v-bind:class="[ active, isActive ? 'div_circle': 'div-box']">背
    景色/圆</div>
    <div v-bind:class="[ {active:true}, isActive ? 'div_circle': 'div-
    box']">背景色/圆</div>
  </div>
</template>

<script>
export default {
  name: 'HelloWorld',
  props: {
    msg: String
  },
  data(){
    return {
      isActive:true,
      isCircle:true,
      active:'active',
    }
  },
}
</script>

<!-- 添加 "scoped" 属性，可以限制CSS只在该组件起效果 -->
<style scoped>
  .active{
    background-color: #adadda;
  }

```

```

    .div_circle{
        width: 200px;
        height: 200px;
        border-radius: 50%;
        border: 2px solid #D19A66;
        line-height: 200px;
        text-align: center;
    }
    .div_box{
        width: 200px;
        height: 200px;
        border: 2px dashed #D3869B;
        line-height: 200px;
        text-align: center;
    }
</style>

```

2. 绑定内联样式

1. 对象语法

`v-bind:style` 的对象语法十分直观——看着非常像 CSS，但其实是一个 JavaScript 对象。CSS 属性名可以用驼峰式 (camelCase) 或短横线分隔 (kebab-case，记得用引号括起来) 来命名：

```

<div v-bind:style="{ color: activeColor, fontSize: fontSize + 'px' }">
</div>
<!--
    data: {
        activeColor: 'red',
        fontSize: 30
    }
-->

<!-- 绑定到一个样式对象-->
<div v-bind:style="styleObject"></div>
<!--
    data: {
        styleObject: {
            color: 'red',
            fontSize: '13px'
        }
    }
-->

```

2. 数组语法

`v-bind:style` 的数组语法可以将多个样式对象应用到同一个元素上

```

<template>
  <div class="hello">
    <div v-bind:class="[ active, 'div_box']">样式绑定——背景色/方形</div>
    <div v-bind:class="[ active, isActive ? 'div_circle': 'div-box']">样
    式绑定——背景色/圆形</div>
  </div>
</template>

<script>

```

```

export default {
  name: 'HelloWorld',
  props: {
    msg: String
  },
  data(){
    return {
      isActive:true,
      isCircle:true,
      active:'active',
    }
  },
}
</script>

<!-- 添加 "scoped" 属性，可以限制CSS只在该组件起效果 -->
<style scoped>
  .active{
    background-color: #adadda;
  }
  .div_circle{
    width: 200px;
    height: 200px;
    border-radius: 50%;
    border: 2px solid #D19A66;
    line-height: 200px;
    text-align: center;
  }
  .div_box{
    width: 200px;
    height: 200px;
    border: 2px dashed #D3869B;
    line-height: 200px;
    text-align: center;
  }
</style>

```

9. 条件指令

1. v-if 指令

`v-if` 指令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真值的时候被渲染。

```

<h1 v-if="awesome">Vue is awesome!</h1>

<!-- v-else 添加一个“else 块” -->
<!-- v-else 元素必须紧跟在带 v-if 或者 v-else-if 的元素的后面，否则它将被不会被识别 -->
<h1 v-if="awesome">Vue is awesome!</h1>
<h1 v-else>Oh no</h1>

<!-- v-else-if 添加一个“else if 块” -->
<!-- 类似于 v-else, v-else-if 也必须紧跟在带 v-if 或者 v-else-if 的元素之后 -->
<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>

```

2. v-show 指令

另一个用于根据条件展示元素的选项是 `v-show` 指令。用法大致一样，不同的是带有 `v-show` 的元素始终会被渲染并保留在 DOM 中。`v-show` **只是简单地切换元素的 CSS `display` 属性**。

```

<!-- v-show 不支持 <template> 元素，也不支持 v-else -->
<h1 v-show="ok">Hello!</h1>

```

3. v-if vs v-show

`v-if` 是“真正”的条件渲染，因为它会确保在切换过程中条件块内的事件监听器和子组件适当地被销毁和重建。

`v-if` 也是**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。

相比之下，`v-show` 就简单得多——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。

一般来说，`v-if` 有更高的切换开销，而 `v-show` 有更高的初始渲染开销。因此，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。

10. 遍历指令

我们可以用 `v-for` 指令基于一个数组来渲染一个列表。`v-for` 指令需要使用 `item in items` 形式的特殊语法，其中 `items` 是**源数据数组**，而 `item` 则是被迭代的数组元素的**别名**。

```

<ul>
  for
  <li v-for="item in items" :key="item.message">
    {{ item.message }}
  </li>
</ul>

```

```

export default {
  name: 'app',
  data(){
    return {
      items: [
        { message: 'Foo' },
        { message: 'Bar' }
      ]
    }
  },
}

```

`v-for` 还支持一个可选的第二个参数，即当前项的索引

```

<ul>
  <li v-for="(item, index) in items">
    {{ parentMessage }} - {{ index }} - {{ item.message }}
  </li>
</ul>

```

```

export default {
  name: 'app',
  data(){
    return {
      parentMessage: 'Parent',
      items: [
        { message: 'Foo' },
        { message: 'Bar' }
      ]
    }
  },
}

```

也可以用 `of` 替代 `in` 作为分隔符，因为它更接近 JavaScript 迭代器的语法

```

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

```

在v-for里使用对象

```

<!-- 也可以用 `v-for` 来遍历一个对象的 property -->
<ul class="demo">
  <li v-for="value in object">
    {{ value }}
  </li>
</ul>

<!-- 也可以提供第二个的参数为 property 名称（也就是键名） -->
<div v-for="(value, name) in object">
  {{ name }}: {{ value }}
</div>

<!-- 还可以用第三个参数作为索引 -->
<div v-for="(value, name, index) in object">
  {{ index }}. {{ name }}: {{ value }}
</div>

```



```
export default {
  name: 'app',
  data(){
    return {
      object: {
        title: 'How to do lists in vue',
        author: 'Jane Doe',
        publishedAt: '2016-04-10'
      }
    }
  },
}
```

注意：在遍历对象时，会按 `Object.keys()` 的结果遍历，但是**不能**保证它的结果在不同的 JavaScript 引擎下都一致。

当 Vue 正在更新使用 `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，而是就地更新每个元素，并且确保它们在每个索引位置正确渲染。

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，你需要为每项提供一个唯一 `key` 属性

```
<!-- 不要使用对象或数组之类的非基本类型值作为 v-for 的 key。请用字符串或数值类型的值。 -->
<div v-for="item in items" v-bind:key="item.id">
  <!-- 内容 -->
</div>
```

11. 事件指令

可以用 `v-on` 指令监听 DOM 事件，并在触发时运行一些 JavaScript 代码。

```
<div>
  <button v-on:click="counter += 1">Add 1</button>
  <p>The button above has been clicked {{ counter }} times.</p>
</div>
```

```
export default {
  name: 'app',
  data(){
    return {
      counter: 0
    }
  },
}
```

1. 事件处理方法

许多事件处理逻辑会更为复杂，所以直接把 JavaScript 代码写在 `v-on` 指令中是不可行的。因此 `v-on` 还可以接收一个需要调用的方法名称。

```

<div>
  <!-- `greet` 是在下面定义的方法名 -->
  <button v-on:click="greet">Greet</button>
</div>

```

```

export default {
  name: 'app',
  data(){
    return {
      name: 'vue.js'
    }
  },
  // 在 `methods` 对象中定义方法
  methods: {
    greet: function (event) {
      // `this` 在方法里指向当前 vue 实例
      alert('Hello ' + this.name + '!')
      // `event` 是原生 DOM 事件
      if (event) {
        alert(event.target.tagName)
      }
    }
  }
}

```

2. 内联处理器中的方法

```

<div>
  <button v-on:click="say('hi')">Say hi</button>
  <button v-on:click="say('what')">Say what</button>
</div>

```

```

export default {
  name: 'app',
  methods: {
    say: function (message) {
      alert(message)
    }
  }
}

```

有时也需要在内联语句处理器中访问原始的 DOM 事件。可以用特殊变量 `$event` 把它传入方法

```

<button v-on:click="warn('Form cannot be submitted yet.', $event)">
  Submit
</button>

```

```
// ...
methods: {
  warn: function (message, event) {
    // 现在我们可以访问原生事件对象
    if (event) {
      console.log(event);
      event.preventDefault()
    }
    alert(message)
  }
}
```

3. 事件修饰符

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

为了解决这个问题，Vue.js 为 `v-on` 提供了**事件修饰符**。修饰符是由点开头的指令后缀来表示的。

```
<!-- .stop -->
<!-- .prevent -->
<!-- .capture -->
<!-- .self -->
<!-- .once -->
<!-- .passive -->

<!-- 阻止单击事件继续传播 -->
<a v-on:click.stop="doThis"></a>

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

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

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

<!-- 添加事件监听器时使用事件捕获模式 -->
<!-- 即内部元素触发的事件先在此处理，后才交由内部元素进行处理 -->
<div v-on:click.capture="doThis">...</div>

<!-- 只当在 event.target 是当前元素自身时触发处理函数 -->
<!-- 即事件不是从内部元素触发的 -->
<div v-on:click.self="doThat">...</div>
```

使用修饰符时，顺序很重要；相应的代码会以同样的顺序产生。因此，用 `v-on:click.prevent.self` 会阻止所有的点击，而 `v-on:click.self.prevent` 只会阻止对元素自身的点击。

4. 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">
```

12. 双向数据绑定

`v-model` 指令在表单控件元素（`select`、`textarea`、`input` 等）上创建双向数据绑定

特点：

1. 根据控件类型自动选取正确的方法来更新元素

```
<!-- 文本 -->
<input v-model="message" placeholder="edit me">
<!-- 本质 -->
<input :value="message" @input="message=$event.target.value">
<p>Message is: {{ message }}</p>

<!-- 多行文本 -->
<textarea v-model="message" placeholder="add multiple lines"></textarea>

<!-- 复选框 -->
<!-- 单个复选框，绑定布尔值 -->
<input type="checkbox" id="checkbox" v-model="checked">
<label for="checkbox">{{ checked }}</label>
<!-- 多个复选框，绑定到同一数组 -->
<input type="checkbox" id="jack" value="Jack" v-model="checkedNames">
<label for="jack">Jack</label>
<input type="checkbox" id="john" value="John" v-model="checkedNames">
<label for="john">John</label>

<!-- 单选按钮 -->
<!-- v-model上绑定同一值 会让单选框互斥，可以不用name属性 -->
<div id="example">
  <input type="radio" id="one" value="One" v-model="picked">
  <label for="one">One</label>
  <br>
  <input type="radio" id="two" value="Two" v-model="picked">
  <label for="two">Two</label>
  <br>
  <span>Picked: {{ picked }}</span>
</div>

<!-- 选择框 -->
<!-- option的value值可以为对象-->
<!-- 单选（绑定值） -->
<div id="example">
  <select v-model="selected">
    <option disabled value="">请选择</option>
    <option>A</option>
    <option>B</option>
    <option>C</option>
  </select>
  <span>Selected: {{ selected }}</span>
</div>
<!-- 多选（绑定数组） -->
<div id="example">
  <select v-model="selected" multiple style="width: 50px;">
```

```

    <option>A</option>
    <option>B</option>
    <option>C</option>
  </select>
  <br>
  <span>Selected: {{ selected }}</span>
</div>

```

2. `v-model` 在内部为不同的输入元素使用不同的属性并抛出不同的事件

1. `text`和`textarea`元素使用`value`属性和`input`事件
2. `checkbox`和`radio`使用`checked`属性和`change`事件
3. `select`使用`value`属性和`change`事件

注意:

1. `v-model` 会忽略表单控件元素自带的属性设定的默认值, 如`selected`、`value`、`checked`

13. 组件基础

```

<div id="components-demo">
  <button-counter></button-counter>
</div>

```

这里定义一个名为 `button-counter` 的新组件, 可以用来复用该组件的内容

```

<!-- 定义一个ButtonCounter.vue文件 -->
<template>
  <button v-on:click="count++">You clicked me {{ count }} times.</button>
</template>

<script>
  export default {
    name: 'ButtonCounter',
    data(){
      return {
        count:0,
      }
    }
  }
</script>

```

1. 组件的复用

组件可以进行任意次数的复用

```

<!-- 每用一次组件, 就会有一个它的新实例被创建 所以每个组件都会各自独立维护它的 count -->
<div id="components-demo">
  <button-counter></button-counter>
  <button-counter></button-counter>
  <button-counter></button-counter>
</div>

```

注意: 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

2. 组件的注册

有两种，一种是局部注册，一种是全局注册

```
<!-- 局部注册 -->
<div id="components-demo">
  <button-counter></button-counter>
</div>

<script>
  import ButtonCounter from './components/ButtonCounter.vue'
  export default {
    name: 'app',
    components: {
      ButtonCounter
    }
  }
</script>

<!-- 全局注册 -->
<!-- 在我们的vue项目中的main.js文件中 -->
Vue.component('组件名称', '组件内容')
```

3. 父向子组件传值

父组件是通过Prop给子组件传值的，Prop是在子组件上注册的一些自定义的属性。当一个值传给一个自定义属性时，它就变成了那个组件实例的一个属性（变量）。通过props选项定义组件可接受的prop列表。

```
<!-- 定义一个blogPost组件，该组件含有其父亲传过来的属性title -->
<template>
  <h3>{{ title }}</h3>
</template>

<script>
  export default {
    name: 'BlogPost',
    props: ['title'],
  }
</script>
```

```
<!-- 自定义的prop在父组件中的使用 -->
<blog-post title="My journey with Vue"></blog-post>
<blog-post title="Blogging with Vue"></blog-post>
<blog-post title="why Vue is so fun"></blog-post>
```

```
<!-- 父组件 -->
<!-- 典型应用中，动态给prop传值 -->
<blog-post
  v-for="post in posts"
  v-bind:key="post.id"
  v-bind:title="post.title">
</blog-post>

<script>
  import BlogPost from './components/BlogPost.vue'
  export default {
```

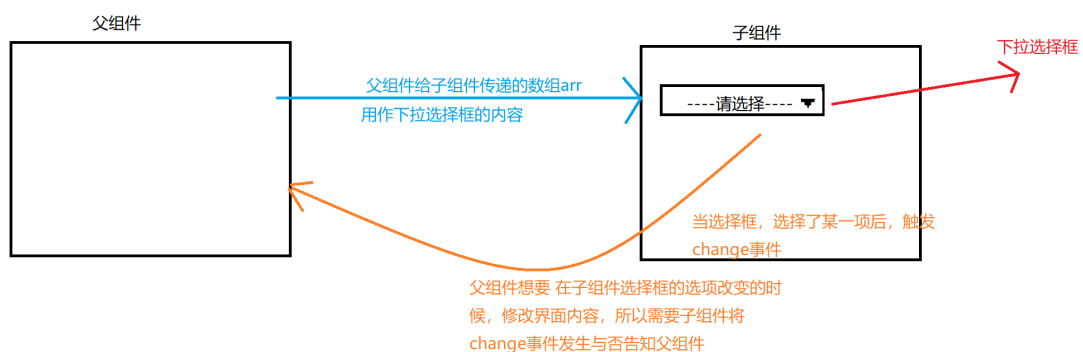
```

    name: 'parent',
    components:{
      BlogPost,
    },
    data(){
      return {
        posts: [
          { id: 1, title: 'My journey with vue' },
          { id: 2, title: 'Blogging with vue' },
          { id: 3, title: 'Why vue is so fun' }
        ]
      }
    }
  }
</script>

```

注意:

1. 一个组件默认可以拥有任意数量的 prop，任何值都可以传递给任何 prop。
 2. 在组件实例中访问这个prop值，就像访问 `data` 中的值一样
 3. **每个组件必须只有一个根元素**，即组件的最顶层只能包含一个html标签元素
4. 监听子组件事件



在子组件中使用`vm.$emit(methodName,...args)`方法，来完成子组件告知父组件（暴露一个事件接口，父组件可以监听）

```

<!-- 父组件 -->
<template>
  <div id="app">
    <m-select :arr="arr" @selected="getSelected"></m-select>
    <div>选项改变次数: {{count}}</div>
  </div>
</template>

<script>
import MSelect from './components/MSelect.vue'
export default {
  name: 'app',
  data() {
    return{
      arr:[
        {label:'选项1',value:'1'},
        {label:'选项2',value:'2'},
        {label:'选项3',value:'3'},
      ],
      count:0,

```

```

    }
  },
  components: {
    MSelect
  },
  methods: {
    getSelected(val) {
      this.count++;
    }
  }
}
</script>

<style>
</style>

```

```

<!-- 子组件 -->
<template>
  <div>
    <select @change="selected">
      <option v-for="item in arr" :key="item.value" :value="item.value">
        {{item.label}}</option>
      </select>
    </div>
  </template>

<script>
  export default {
    name: 'MSelect',
    props: {
      arr: {
        type: Array,
        default: []
      },
    },
    methods: {
      selected(val) {
        this.$emit('selected', val.target.options[val.target.options.selectedIndex].value);
      }
    }
  }
</script>

<style>
</style>

```

扩展:

1. model属性实现自定义组件的双向数据绑定

实现双向绑定的关键代码就是定义**model属性中的prop和event**，v-model中的值传递给model属性中的prop对应的变量。然后在自定义组件中选择一个要传递出去值的基本组件，通过\$emit发送event事件并传递一个结果值，这样外部的v-model就收到了传出的值，因此就实现了双向传递。

14. 插槽

插槽 (Slot) 是Vue提出来的一个概念，正如名字一样，插槽用于决定将所携带的内容，插入到指定的某个位置，从而使模板分块，具有模块化的特质和更大的重用性。**插槽显不显示、怎样显示是由父组件来控制的，而插槽在哪里显示就由子组件来进行控制**

在实际使用中通过 `<slot>` 标签来实现插槽。Slot 是组件内部的占位符，用户可以使用自己的标记来填充。

1. 匿名插槽

```
<!-- 父组件 -->
<!-- 在父组件引用的子组件中写入想要显示的内容（可以使用标签，也可以不用） -->
<template>
  <div>
    我是父组件
    <my-slot-one>
      <p style="color:red">我是父组件插槽内容</p>
    </my-slot-one>
  </div>
</template>
```

```
<!-- 子组件 my-slot-one -->
<!-- 在子组件中写入slot，slot所在的位置就是父组件要显示的内容 -->
<template>
  <div>
    <div>我是子组件</div>
    <slot></slot>
  </div>
</template>
```

2. 具名插槽

```
<!-- 子组件 my-slot-two -->
<!-- 在子组件中定义了三个slot标签，其中有两个分别添加了name属性header和footer -->
<template>
  <div>
    <div>我是子组件-具名插槽</div>
    <slot name="header"></slot>
    <slot></slot>
    <slot name="footer"></slot>
  </div>
</template>
```

```

<!-- 父组件 -->
<template>
  <div>
    我是父组件
    <my-slot-two>
      <p>啦啦啦，啦啦啦，我是卖报的小行家</p>
      <template slot="header">
        <p>我是name为header的slot</p>
      </template>
      <p slot="footer">我是name为footer的slot</p>
    </my-slot-two>
  </div>
</template>

```

在父组件中使用template并写入对应的slot值来指定该内容在子组件中现实的位置（当然也不用必须写到template），没有对应值的其他内容会被放到子组件中没有添加name属性的slot中

注意：在插槽中也可以定义默认值，当父组件中没有写入内容的时候，会显示子组件的默认内容，当父组件中写入了内容时会替换子组件的默认内容。

3. 作用域插槽

我们在父组件中对子组件的插槽进行添加内容时，可能会使用到子组件里面的一些数据。这时就可以使用作用域插槽，来提升子组件中数据的作用域，暴露出来以便父组件使用。

```

<!-- 子组件 my-slot-three -->
<!-- 在子组件的slot标签上绑定需要的值 -->
<template>
  <div>
    我是作用域插槽的子组件
    <slot :data="user"></slot>
  </div>
</template>

<script>
export default {
  name: 'myslotthree',
  data () {
    return {
      user: [
        {name: 'Jack', sex: 'boy'},
        {name: 'Jone', sex: 'girl'},
        {name: 'Tom', sex: 'boy'}
      ]
    }
  }
}
</script>

```

```

<!-- 父组件 -->
<!-- 在父组件上使用slot-scope属性，user.data就是子组件传过来的值 -->
<template>
  <div>
    我是作用域插槽
    <my-slot-three>
      <template slot-scope="user">
        <div v-for="item in user.data" :key="item.id">

```

```

        {{item}}
      </div>
    </template>
  </my-slot-three>
</div>
</template>

```

15. Vue Router的简单运用

前端中所说的路由是根据不同的用户事件，显示不同的页面内容，本质上是用户事件与事件处理函数之间的对应关系。前端路由负责事件监听，触发事件后，通过事件函数渲染不同的内容。

1. 声明式路由

`<router-link>` 是 vue 中提供的标签，默认会被渲染为 a 标签，to 属性默认会被渲染为 a 标签的 href 属性。

`<router-view>` 通过路由规则匹配到的组件，将会被渲染到 router-view 所在的位置 展示在页面上

1) 不包含参数

```

<!-- 创建一个新的vue界面 Home.vue -->
<template>
  <div class="home">
    
    <HelloWorld msg="Welcome to Your Vue.js App"/>
  </div>
</template>

<script>
// @ is an alias to /src
import HelloWorld from '@components/HelloWorld.vue'

export default {
  name: 'home',
  components: {
    HelloWorld
  }
}
</script>

```

```

<!-- 创建一个新的vue界面 About.vue -->
<template>
  <div class="about">
    <h1>This is an about page</h1>
  </div>
</template>

```

```

// 修改我们的router下的js文件，即配置路由信息的js文件
import Vue from 'vue'
import Router from 'vue-router'
import Home from '../views/Home.vue'

Vue.use(Router)

export default new Router({

```

```

mode: 'history',
base: process.env.BASE_URL,
routes: [
  {
    path: '/',
    name: 'home',
    component: Home
  },
  {
    path: '/about',
    name: 'about',
    // route level code-splitting
    // this generates a separate chunk (about.[hash].js) for this route
    // which is lazy-loaded when the route is visited.
    component: () => import('../views/About.vue')
  }
]
})

```

```

<!-- 在App.vue中使用router-link和router-view的方式声明使用路由 -->
<template>
  <div id="app">
    <div id="nav">
      <router-link to="/">Home</router-link> |
      <router-link to="/about">About</router-link>
    </div>
    <router-view/>
  </div>
</template>

<style scoped>
  #app{
    font-family: 'Avenir', Helvetica, Arial, sans-serif;
    -webkit-font-smoothing: antialiased;
    -moz-osx-font-smoothing: grayscale;
    text-align: center;
    color: #2c3e50;
    margin-top: 60px;
  }
</style>

```

2) 包含参数

当我们进行路由跳转的时候，后面的界面经常会需要使用到前面界面传递过来的一些数据。这时就需要在路由跳转中带入参数。可以在 `vue-router` 的路由路径中使用“动态路径参数”(dynamic segment) 来达到这个效果。

```

<!-- 新建一个vue界面 User.vue -->
<template>
  <div>User</div>
</template>

```

```

// 修改router配置文件
import Vue from 'vue'
import Router from 'vue-router'
import Home from '../views/Home.vue'

```

```
import User from '../views/User.vue'

Vue.use(Router)

export default new Router({
  routes: [
    {
      path: '/',
      name: 'home',
      component: Home
    },
    {
      path: '/about',
      name: 'about',
      component: () => import('../views/About.vue')
    },
    {
      // 动态路径参数，以冒号开头
      path: '/user/:id',
      name: 'user',
      component: User,
    }
  ]
})
```

```
<!-- 在App.vue中使用router-link和router-view的方式声明使用路由 -->
<template>
  <div id="app">
    <div id="nav">
      <router-link to="/">Home</router-link> |
      <router-link to="/about">About</router-link> |
      <router-link to="/user/lily">User1</router-link> |
      <router-link to="/user/tom">User1</router-link>
    </div>
    <router-view/>
  </div>
</template>

<style scoped>
  #app{
    font-family: 'Avenir', Helvetica, Arial, sans-serif;
    -webkit-font-smoothing: antialiased;
    -moz-osx-font-smoothing: grayscale;
    text-align: center;
    color: #2c3e50;
    margin-top: 60px;
  }
</style>
```

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.$route.params`，可以在每个组件内使用。因此如果我们想要在User.vue界面中获取到这个参数时，可以使用`this.$route.params.id`来获取

```

<!-- 使用参数的User.vue -->
<template>
  <div>User {{$route.params.id}}</div>
</template>

```

3) 路由嵌套

实际生活中的应用界面，通常由多层嵌套的组件组合而成。同样地，URL 中各段动态路径也按某种结构对应嵌套的各层组件

```

<!-- 新建一个vue界面 Sign.vue -->
<template>
  <div>Sign</div>
</template>

```

```

// 修改router的配置文件
import Vue from 'vue'
import Router from 'vue-router'
import Home from '../views/Home.vue'
import User from '../views/User.vue'
import Sign from '../views/Sign.vue'

Vue.use(Router)

export default new Router({
  mode: 'history',
  base: process.env.BASE_URL,
  routes: [{
    path: '/',
    name: 'home',
    component: Home
  },
  {
    path: '/about',
    name: 'about',
    // route level code-splitting
    // this generates a separate chunk (about.[hash].js) for this
    // route
    // which is lazy-loaded when the route is visited.
    component: () => import( /* webpackChunkName: "about" */
      '../views/About.vue')
  },
  {
    // 动态路径参数，以冒号开头
    path: '/user/:id',
    name: 'user',
    component: User,
    children: [
      {
        // 当 /user/:id/sign 匹配成功，
        // Sign 会被渲染在 User 的 <router-view> 中
        // 要注意，以 / 开头的嵌套路径会被当作根路径。
        path: 'sign',
        component: Sign
      }
    ]
  }
]

```

```
    }  
  ]  
})
```

```
<!-- 修改User.vue, 在里面嵌套使用路由 -->  
<template>  
  <div>  
    <h2>User {{$route.params.id}}</h2>  
    <router-link :to="'/user/'+$route.params.id+'/sign'">路由的嵌套</router-link>  
    <router-view></router-view>  
  </div>  
</template>
```

\$router: 是路由操作对象, 只写对象

\$route: 路由信息对象, 只读对象

2. 程式化路由

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。

1) router.push(location,onComplete?,onAbort?)

在 Vue 实例内部, 你可以通过 `$router` 访问路由实例。因此你可以调用 `this.$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>` 等同于调用 `router.push(...)`。

```
// 字符串  
router.push('home')  
  
// 对象  
router.push({ path: 'home' })  
  
// 命名的路由  
router.push({ name: 'user', params: { userId: '123' } })  
  
// 带查询参数, 变成 /register?plan=private  
router.push({ path: 'user/tom', query: { msg: 'message' } })
```

如果提供了 `path`, `params` 会被忽略。如果需要带参数, 需要提供路由的 `name` 或手写完整的带有参数的 `path`

注意:

1. `params`是路由的一部分, 必须要在路由后面添加参数名。 `query`是拼接在url后面的参数, 没有也没关系。
2. `params`一旦设置在路由, `params`就是路由的一部分, 如果这个路由有`params`传参, 但是在跳转的时候没有传这个参数, 会导致跳转失败或者页面会没有内容

```
const userId = '123'  
router.push({ name: 'user', params: { userId }}) // -> /user/123  
router.push({ path: `/user/${userId}` }) // -> /user/123  
// 这里的 params 不生效  
router.push({ path: '/user', params: { userId }}) // -> /user
```

如果目的地和当前路由相同，只有参数发生了改变 (比如从一个用户资料到另一个 `/users/1` -> `/users/2`)，这时实质上是没有进行路由的跳转，只是通过获取参数，动态渲染了用户界面。

2) `router.replace(location,onComplete?,onAbort?)`

跟 `router.push` 很像，唯一的不同就是，**它不会向 history 添加新记录**，而是跟它的方法名一样——替换掉当前的 history 记录

如果想在声明式跳转路由中使用 `replace`，只需要在 `<router-link>` 标签上面加上 `replace`

```
<router-link to="..." replace>
```

```
// 字符串  
router.replace('home')  
  
// 对象  
router.replace({ path: 'home' })  
  
// 命名的路由  
router.replace({ name: 'user', params: { userId: '123' }})  
  
// 带查询参数，变成 /register?plan=private  
router.replace({ path: 'user/lily', query: { msg: 'message' }})
```