Vue中的核心知识点。

1、基本使用

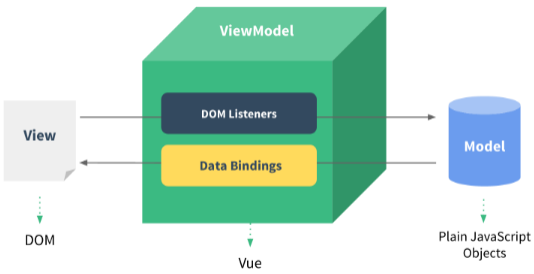
Vue的核心理念是数据驱动的理念，所谓的数据驱动的理念：当数据发生变化的时候，用户界面也会发生相应的变化，开发者并不需要手动的去修改dom。简单的理解：就是vue.js帮我们封装了数据和dom对象操作的映射，我们只需要关心数据的逻辑处理，数据的变化就能够自然地通知页面进行页面的重新渲染。这样做给我们带来的好处就是，我们不需要在代码中去频繁的操作dom了，这样提高了开发的效率，同时也避免了在操作Dom的时候出现的错误。

Vue.js的数据驱动是通过MVVM这种框架来实现的，MVVM框架主要包含三部分：Model,View,ViewMode

Model:指的是数据部分，对应到前端就是JavaScript对象。View:指的就是视图部分

ViewModel: 就是连接视图与数据的中间件(中间桥梁)

MVVM框架的作用:



数据(Model)和视图(View)是不能直接通讯的，而是需要通过ViewModel来实现双方的通讯。当数据(Model)变化的时候，ViewModel能够监听到这种变化，并及时通知View视图做出修改。同样的，当页面有事件触发的时候，ViewModel也能够监听到事件，并通知数据(Model)进行响应。所以ViewModel就相当于一个观察者，监控着双方的动作，并及时通知对方进行相应的操作。

简单的理解就是：MVVM实现了将业务(数据)与视图进行分离的功能。

注意：MVVM框架的三要素：响应式，模板引擎，渲染。

2、模板语法

```
    <div id="app">
      <h2 :title="msg">
        {{msg}}
      </h2>
    </div>
```

属性绑定，为了避免闪烁的问题，也就是最开始的时候，出现{{msg}}的情况，可以使用如下的绑定方式。

```
<div id="app">
    <h2 :title="msg">
      <span v-text="msg"></span>
    </h2>
</div>
```

3、 列表渲染

我们可以使用`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`形式的语法。其中`items`是源数组，而`item`则是被迭代的数组元素的别名。

```
      <ul>
        <li v-for="(item,index) in users" :key="item.id">
          编号：{{item.id}} 姓名:{{item.name}}---索引:{{index}}
        </li>
      </ul>
```

注意：为了能够保证列表渲染的性能，我们需要给`v-for`添加`key`属性。`key`值必须唯一，而且不能使用`index`与`random`作为`key`的值。关于这一点是与虚拟`DOM`算法密切相关的。

4、v-model

v-model指令用来双向数据绑定：就是model和view中的值进行同步变化
v-model只能在input/textarea/selet 也就是表单元素

```
      <input type="text" v-model="userName" />
```

5、v-on

监听`dom`的事件可以通过`v-on`指令完成，建议以后用简写的形式。也有带参数的形式。也可以绑定键盘的事件。

```
<div id="app">
      <span>{{name}}</span>
      <button @click="changeName">更换姓名</button>
      <button @click="changeNameByArg('laowang')">带参数的情况</button>
      <input type="text" @keydown.enter="changeUserName" v-model="name" />
    </div>  <!-- 按键的修饰符：`.enter`-->
```

```
        methods: {
          changeName() {
            this.name = "itcast";
          },
          changeNameByArg(userName) {
            this.name = userName;
          },
          changeUserName() {
            console.log(this.name);
          },
        },
```

6、Class与Style绑定

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。

`Class`的绑定：

```
          :class="{actived:true}"
```

现在有一个需求，就是当鼠标移动到列表项上的时候，更改对应的class。

```
        :class="{actived:selectItem===item}"   @mousemove="selectItem=item"
```

`Style`的绑定：可以看到通过绑定`style`的方式来处理样式是非常麻烦的

```
        :style="{backgroundColor:selectItem===item?'#dddddd':'transparent'}"
        @mousemove="selectItem=item"
```

7、条件渲染

v-if和v-show指令可以用来控制元素的显示和隐藏。

`v-show` 是通过`css`属性`display`控制元素显示，元素总是存在的。

`v-if` 通过控制`dom`来控制元素的显示和隐藏,如果一开始条件为`false`,元素是不存在的。

如果需要频繁的控制元素的显示与隐藏，建议使用`v-show`. 从而避免大量`DOM`操作，提高性能。

如果某个元素满足条件后，渲染到页面中，并且以后变化比较少，可以使用`v-if`

8、计算属性

计算属性出现的目的是解决模板中放入过多的逻辑会让模板过重且难以维护的问题.

计算属性是根据data中已有的属性，计算得到一个新的属性.

```
        <div>全名：{{firstName + lastName}}</div>
        <div>全名：{{fullName}}</div>
            computed: {  // 创建计算属性通过computed关键字，它是一个对象
                // 这里fullName就是一个计算属性，它是一个函数，但这个函数可以当成属性来使用
                fullName() {
                    return this.firstName + this.lastName
                }
            }
```

```
       <p>总人数：{{users.length+"个"}}</p>
```

这里在模板中做了运算（在这里做了字符串拼接，虽然计算简单，但是最好还是通过计算属性来完成），为了防止在模板中放入过多的逻辑计算，这里可以使用计算属性来解决。

```
       <p>总人数：{{total}}</p>
     computed: {
              total() {
                // 计算属性是有缓存性：如果值没有发生变化，则页面不会重新渲染
                return this.users.length + "个";
              },
            }
```

可以看到使用计算属性，让界面变得更加的简洁。

使用计算属性还有一个好处：调用methods里的方法也能实现和计算属性一样的效果，既然使用methods就可以实现，那为什么还需要计算属性呢？原因就是计算属性是基于它的依赖缓存的（所依赖的还是`data`中的数据）。一个计算属性所依赖的数据发生变化时，才会重新取值。也就是说：只要相关依赖没有改变，对此访问计算属性得到的是之前缓 存的结果，不会多次执行。所以说，在进行大量耗时计算的时候，建议使用计算属性来完成，通过计算属性可以提升性能。

```
<p>总人数：{{getTotal()}}</p>
<p>总人数：{{getTotal()}}</p>
 methods: {
         getTotal: function () {
           console.log("methods");
           return this.users.length + "个";
         },
       },
```

在上面的代码中，调用了两次`getTotal`方法。

9、侦听器

侦听器就是侦听`data`中的数据变化，如果数据一旦发生变化就通知侦听器所绑定方法，来执行相应的操作。

```
<p>总人数：{{totalCount}}</p>
```

```
  watch: {
         users: {
           immediate: true, //立即执行，表示的就是在初始化绑定的时候，也会去执行侦听器
           handler(newValue, oldValue) {
             this.totalCount = newValue.length + "个人";
           },
         },
       }
```

计算属性与侦听器总结：

第一点：语境上的差异：

侦听器`watch`适合一个值发生了变化，对应的要做一些其它的事情，适合一个值影响多个值的情形。

计算属性`computed`:一个值由其它值得来，其它值发生变化，对应的值也会变化，适合多个值影响一个值的情形。

第二点：计算属性有缓存性。由于这个特点，在实际的应用中，能用计算属性的，会先考虑先使用计算属性。

第三点：侦听器选项提供了更加通用的方法，适合执行异步操作或者较大开销操作。

10、生命周期简介

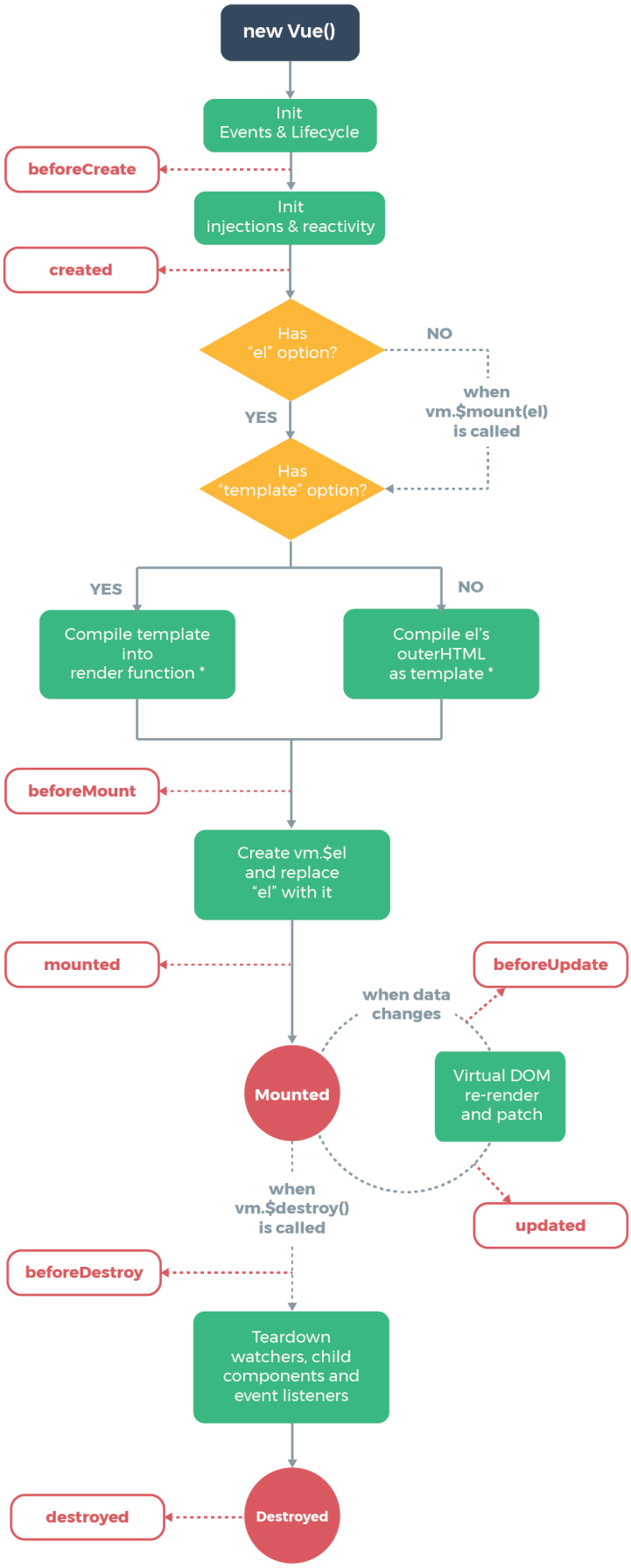
每个`Vue`实例在被创建时都要经过一系列的初始化过程，例如：需要设置数据的监听，编译模板，将实例挂载到`DOM`上，并且在数据变化时更新`DOM`等，这些过程统称为`Vue`实例的`生命周期`。同时在这个过程中也会运行一些叫做**生命周期钩子**的函数，这给了用户在不同阶段添加自己的代码的机会。

11、生命周期探讨

`Vue`实例的生命周期，主要分为三个阶段，分别为

- 挂载(初始化相关属性,例如`watch`属性，`method`属性)：beforeCreate created beforeMount mounted
- 更新(元素或组件的变更操作)：beforeUpdate updated
- 销毁（销毁相关属性）：beforeDestroy destroyed

```
关于Vue的生命周期，下列哪项是不正确的？(b)[单选题]
A、Vue 实例从创建到销毁的过程，就是生命周期。
B、页面首次加载会触发beforeCreate, created, beforeMount, mounted, beforeUpdate, updated。
C、created表示完成数据观测，属性和方法的运算，初始化事件，$el属性还没有显示出来。
D、DOM渲染在mounted中就已经完成了。
```



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

`beforeCreate`：`Vue`实例初始化之后，以及事件初始化，以及组件的父子关系确定后执行该钩子函数，一般在开发中很少使用。

`created`：在调用该方法之前，初始化会被使用到的状态，状态包括`props`,`methods`,`data`,`computed`,`watch`。而且会实现对`data`中属性的监听，也就是在`created`的时候数据已经和`data`属性进行了绑定。(放在`data`中的属性当值发生改变的时候，视图也会改变)。同时也会对传递到组件中的数据进行校验。所以在执行`created`的时候，所有的状态都初始化完成，也可在该阶段发送异步的`ajax`请求，获取数据。但是，在`created`方法中，是无法获取到对应的的`$el`选项,也就是无法获取`Dom`。

`created`方法执行完毕后，下面会判断对象中有没有`el`选项。如果有，继续执行下面的流程，也就是判断是否有`template`选项，如果没有`el`选项，则停止整个生命周期的流程，直到执行了`vm.$mount(el)`后，才会继续向下执行生命周期的流程。

继续向下就是判断在对象中是否有`template`选项。第一：如果`Vue`实例对象中有`template`参数选项，则将其作为模板编译成`render`函数，来完成渲染。第二：如果没有`template`参数选项，则将外部的`HTML`作为模板编译（`template`），也就是说，`template`参数选项的优先级要比外部的`HTML`高。第三：如果第一条,第二条件都不具备，则报错。

```
<script src="./vue.js"></script>
<div id="app"></div>
<script>
  const vm = new Vue({
    el: "#app",
    template: "<p>Hello {{message}}</p>", //在`Vue`实例中添加`template`的情况
    data: {
      message: "vue",
    },
  });
</script>
```

当模板同时放在`template`参数选项和外部`HTML`中，即添加了`template`属性，也在外部添加了模板内容，但是最终在页面上显示的是`Hello vue`而不是“你好”。就是因为`template`参数的优先级比外部`HTML`的优先级要高。当然在开发中，基本上都是使用外部的`HTML`模板形式，因为更加的灵活。

```
<div id="app">
  <p>你好</p>
</div>
<script>
  const vm = new Vue({
    el: "#app",
    template: "<p>Hello {{message}}</p>",
    data: {
      message: "vue",
    },
  });
</script>
```

为什么先判断`el`选项，然后再判断`template`选项？因为`Vue`需要通过`el`的“选择器”找到对应的`template`，也就是说，`Vue`首先通过`el`参数去查找对应的`template`。如果没有找到`template`参数，则到外部`HTML`中查找，找到后将模板编译成`render`函数（`Vue`的编译实际上就是指`Vue`把模板编译成`render`函数的过程）。

`beforeMount`：在执行该钩子函数的时候，虚拟`DOM`已经创建完成，马上就要渲染了，在这里可以更改`data`中的数据，不会触发`updated`（其实在`created`中也是可以更改数据，也不会触发`updated`函数）。

`mounted`：可看到真实的数据。同时整个组件内容已挂载到页面中了，数据以及真实`DOM`都已经处理好了，可以在这里操作真实`DOM`了。即在`mounted`的时候，页面已被渲染完毕了，在这钩子函数中，可以发送`ajax`请求。

在`updated`之前`beoreUpdate`之后有一个非常重要的操作就是虚拟`DOM`会重新构建，也就是新构建的虚拟`DOM`与上一次的虚拟`DOM`树利用`diff`算法进行对比之后重新渲染。而到了`updated`这个方法，就表示数据已经更新完成，`dom`也重新`render`完成。

如果调用了`vm.$destroy`方法后，就会销毁所有的资源。首先会执行`beforeDestroy`这个钩子函数，在实例销毁前调用，在这一步，实例仍然可用。在该方法中，可以做一些清理的工作，例如：清除定时器等。但是执行到`destroyed`钩子函数的时候，`Vue`实例已经被销毁，所有的事件监听器会被移除，所有的子实例也会被销毁。

12、组件化应用

组件表示页面中的部分功能（包含自己的逻辑与样式），可以组合多个组件实现完整的页面功能。

如何确定应该将哪些部分划分到一个组件中呢？可以将组件当作一种函数或者是对象来考虑（函数的功能是单一的），根据[**单一功能原则**]来判定组件的范围。也就是说，一个组件原则上只能负责一个功能。如果它需要负责更多的功能，这时候就应该考虑将它拆分成更小的组件。

组件特点：可复用、维护、可组合。可复用：每个组件都是具有独立功能的，它可以被使用在多个场景中。可组合：一个组件可以和其它的组件一起使用或者可以直接嵌套在另一个组件内部。可维护：每个组件仅仅包含自身的逻辑，更容易被理解和维护。

组件具体的创建过程，第一个参数指定了所创建的组件的名字，第二个参数指定了模板。

注意：组件模板中必须有一个根元素，模板template中只能有一个根节点；组件模板内容可以使用模板字符串，在组件的模板中使用类模板字符串，这样就可以调整对应的格式，例如换行等；组件的名字，如果采用驼峰命令的话，在使用的时候，就要加上“-”，比如组件名字叫indexA，那么在使用的时候就叫index-a。创建的组件是全局组件，可以在其它组件中使用。如果在`componentA`这个组件中使用`HelloWorld`这个组件的时候，可以使用驼峰命名的方式，但是在`<div id="app"></div>`这个普通的标签模板中，必须使用短横线的方式，才能使用组件。

在Vue实例中所使用的选项，在组件中都可以使用，但是要注意data在组件中使用时必须是一个函数。

```
<body>
    <div id="app">
        <component-a></component-a> <!-- 组件创建好以后，具体的使用方式 -->
        <index></index>
        <index></index>
        <about></about>
    </div>
    <script>
        Vue.component('componentA', {
            template: "<div>创建一个新的组件</div>"
        })
        Vue.component('index', {
            template: `<div>我是首页的组件</div>`
        })
        Vue.component('about', {
            template: '<div>{{msg}}<button @click="showMsg">单击</button></div>',
            data() {
                return {
                    msg: '大家好'
                }
            },
            methods: {
                showMsg() {
                    this.msg = "关于组件"
                }
            }
        })
        var vm = new Vue({
            el: '#app',
```

```
            data: { }
        })
    </script>
</body>
```

局部组件注册：可以在一个组件中，再次注册另外一个组件，这样就构成了父子关系。可以通过components 来创建对应的子组件。`son`组件是一个局部的组件，那么只能在其注册的父组件中使用。当然，在`father`中定义子组件`son`的时候，直接在其内部构件模板内容，这样如果代码非常多的时候，就不是很直观。所以这里可以将`son`组件，单独的进行定义，然后在`father`组件中进行注册。

```
<script>
        Vue.component('father', {
            template: '<div><p>我是父组件</p><son></son></div>',
            components: {
                // 创建一个子组件
                son: {
                    template: '<p>我是子组件</p>'
                }
            }
        })
        var vm = new Vue({
            el: '#app',
            data: { }
        })
    </script>
```

也可以在（全局）`Vue`实例中，注册对应的局部组件。因为可以将`vue`实例作为一个组件。

```
    <div id="app">
      <component-a></component-a>
      <hello-msg></hello-msg>
    </div>
    <script>
      const son = {
        data() {
          return {
            msg: "Hello 我是子组件",
          };
        },
        template: `<div>{{msg}}</div>`,
      };
      const HelloMsg = {
        data() {
          return {
            msg: "Hello World",
          };
        },
        template: `<div>{{msg}}</div>`,
      };
      Vue.component("ComponentA", {
        template: "<div><son></son></div>",
        components: {
          son: son,
        },
      });
      var vm = new Vue({
        el: "#app",
```

```
        data: {},
        components: {
          "hello-msg": HelloMsg,  //同理，在其他的组件中是无法使用`HelloMsg`组件的
        },
      });
    </script>
```

13、组件通信

常见的组件的通信可以分为三类：父组件向子组件传递数据，子组件向父组件传递数据，兄弟组件的数据传递。

①父组件向子组件传递数据：子组件内部通过 `props` 接收传递过来的值。父组件通过属性将值传递给子组件。

```
Vue.component('menu-item',{
 props:['title'] // props后面跟一个数组，数组中的内容为字符串，这个字符串可以当做属性类使用。
 template:'<div>{{title}}</div>'
})
```

```
<menu-item title="向子组件传递数据"> </menu-item>
<menu-item :title="title"></menu-item> <!--可以使用动态绑定的方式来传值-->
```

`props` 在进行命名的时候，也是有一定的规则的。如果在 `props` 中使用驼峰形式，模板中要用短横线形式。

`props` 可接收各种类型的值：字符串(`String`),数值(`Number`),布尔值(`Boolean`),数组(`Array`),对象(`Object`)

```
Vue.component('menu-item',{
    //在JavaScript中是驼峰形式
    props:['menuTitle'],
    template:'<div>{{menuTitle}}</div>'
})
<!--在html中是短横线方式--->
    <menu-item menu-title="hello world"></menu-item>
```

②子组件向父组件传值：子组件通过自定义事件向父组件传递信息。父组件监听子组件的事件。

```
<button v-on:click='$emit("countSum")'> 计算</button>
<!--子组件传值给父组件要用到$emit()方法，这个方法可传递两个参数，一个是事件名称，一个是需要传递的数据-->
```

```
<menu-item v-on:countSum='sum+=1'></menu-item>
```

③兄弟组件之间数据传递：通过事件总线完成。

A定义父组件并且在父组件中，完成两个兄弟组件的创建。

B创建事件总线：通过事件总线发射一个事件名称和需要传递的数据 。

```
        var eventbus = new Vue()  // 创建一个空的vue实例，作为事件总线
        ......
        eventbus.$emit('tellBroMyName', this.myName)
```

C通过eventbus的$on()方法去监听兄弟节点发射过来的事件

```
       eventbus.$on('tellBroMyName', data => {
          this.mySisterName = data
       } // $on有两个参数，一个是事件名称，一个是函数，该函数的默认值就是传递过来的数据
```

④组件的使用

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

14、组件插槽应用

组件中的插槽，让使用者可以决定组件内部的一些内容到底展示什么，也就是，插槽可以实现父组件向子组件传递模板内容。具有插槽的组件将会有更加强大的拓展性。

将共性抽取到组件中，将不同暴露给插槽，一旦使用了插槽，就相当于预留了空间，空间的内容取决于使用者。

基本使用方式：

第一：确定插槽的位置：在子组件中，通过`<slot>`确定出插槽的位置。

第二：插槽内容：向插槽中传递内容。当然在插槽中也可以添加默认的内容。

```
    <alert-box>程序出现了bug</alert-box>
    <alert-box>程序出现了警告</alert-box>
    <alert-box></alert-box>
  </div>
  <script src="./vue.js"></script>
  <script>
    Vue.component("alert-box", {
      template: `
              <div>
                  <strong>ERROR:</strong>
                  <slot>默认内容</slot>
              </div>
          `,
    });......
```

①具名插槽：有名字的插槽。

```
    <base-layout>  <!--第二：插槽内容-->
      <p slot="header">头部内容</p>
      <p>主要内容1</p>
      <p>主要内容2</p>
      <p slot="footer">底部信息</p>
    </base-layout>
  </div>
  <script src="./vue.js"></script>
  <script>
    Vue.component("base-layout", {
      template: `
              <div>
                  <header>
                      <slot name="header"></slot>  //第一：插槽定义
                   </header>
                   <main>
                      <slot></slot>
                   </main>
                   <footer>
                      <slot name="footer"></slot>
                   </footer>
              </div>
```

```
        `,
    });......
```

在上面的应用中，有一个问题就是，把插槽的名称给了某个`html`标签，例如`p`标签，这样就只能将该标签插入到插槽中。但在实际应用中，有可能需要向插槽中插入大量的内容，这时就需要用到`template`标签。可以给`template`标签添加插槽的名称，并且在`template`标签中嵌入其它多个标签，从而完成布局。

```
    <base-layout>
      <template slot="header">
        <div>标题名称</div>
        <div>标题区域的布局</div>
      </template>
      <div>
        中间内容区域的布局实现
      </div>
      <template slot="footer">
        <div>底部信息</div>
        <div>对底部内容区域进行布局</div>
      </template>
    </base-layout>......
```

②作用域插槽：应用场景：父组件对子组件的内容进行加工处理。模板虽然是在父级作用域（父组件）中渲染的，却能拿到子组件的数据。作用域插槽的作用就是将组件的数据暴露出去。而这么做，给了组件的使用者根据数据定制模板的机会，组件不再是写死成一种特定的结构。

```
    <user-list :list="userList">
      <template slot-scope="slotProps">   <!--固定写法-->
        <strong v-if="slotProps.info.id===2">{{slotProps.info.userName}}</strong>
        <span v-else>{{slotProps.info.userName}}</span>
      </template> <!--从父组件中进行修改。通过父组件来决定子组件中的哪个用户名进行高亮显示。-->
    </user-list>
  </div>
  <script src="./vue.js"></script>
  <script>
    Vue.component("user-list", {
      props: ["list"],
      template: `<div>
              <ul>
                  <li :key="item.id" v-for='item in list'>
                      <slot :info="item">
                          {{item.userName}}
                          </slot>
                      </li>
               </ul>
              </div>`,
    });......
```

作用域插槽案例

```
    <!-- 如果没有传递模板，那么子组件的插槽中只会展示用户名 -->
    <my-list title="用户列表" :content="listData"></my-list>
      <!-- 传递模板 -->
      <my-list title="用户列表2" :content="listData">
        <template slot-scope="scope">
          <img src="./one.png" width="20"/>
          <span>{{scope.item.userName}}</span>
```

```
        </template>
      </my-list>
    </div>
    <script src="./vue.js"></script>
    <script>
      Vue.component("my-list", {
        props: ["title", "content"],
        template: `
                <div class="list">
                    <div class="list-title">
                        {{title}}
                    </div>
                    <div class="list-content">
                   <ul class="list-content">
                            <li v-for="item in content" :key="item.id">
          <!--这里将content中的每一项数据绑定到slot的item变量上，在父组件中就可以获取到item变量-->
                               <slot :item="item">{{item.userName}}</slot>
                            </li>
                        </ul>
                    </div>
                </div>`,
      });.....
```

在使用的 Vue 插件或者是第三方的库中，遇到使用作用域插槽的情况比较典型的就是 element-ui 的 table 组件，它就可以通过添加作用域插槽改变渲染的原始数据。

组件的最大特性就是复用性，而用好插槽能大大提高组件的可复用能力。但是，卡片是在父组件上代替子组件实现的功能，使用插槽无疑是在给父组件页面增加规模，如果全都使用拼装的方式，和不用组件又有什么区别，因此，插槽并不是用的越多越好。插槽是组件最大化利用的一种手段，而不是替代组件的策略，当然也不能替代组件。如果能在组件中实现的模块，或者只需要使用一次 v-else， 或一次 v-else-if， v-else 就能解决的问题，都建议直接在组件中实现。

15、Vue 组件化的理解

定义：组件是可复用的 Vue 实例，准确讲它是 VueComponent 的实例，继承自 Vue。优点：组件化可以增加代码的复用性，可维护性和可测试性。

使用场景：分类：①通用组件：实现最基本的功能，具有通用性，复用性。例如按钮组件，输入框组件，布局组件等。(Element UI 组件库就是属于这种通用的组件) ②业务组件，用于完成具体的业务，具有一定的复用性。例如登录组件，轮播图组件。 ③页面组件，组织应用各部分独立内容，需要时在不同页面组件间切换，例如：商品列表页，详情页组件。

使用组件：定义：Vue.component()， components 选项。分类：有状态组件(有data属性)， functional。

通信：props， $emit()/$on()， provide/inject。内容分发：<slot>， <template>， v-slot。

使用及优化：is， keep-alive， 异步组件。

组件的本质：vue 中的组件经历如下过程 组件配置 => VueComponent 实例 => render() => Virtual DOM => DOM，所以组件的本质是产生虚拟 DOM

16、常用 API 说明

① Vue.set：向响应式对象中添加一个属性，并确保这个新属性同样是响应式的，且会触发视图更新。

使用方法：Vue.set(target,propertyName,value)

```
  batchUpdate() {
            this.users.forEach((c) => {
              //   c.height = this.height;
              Vue.set(c, "height", this.height);
            });
          },
```

② `Vue.delete`：删除对象的属性，如果对象是响应式的，确保删除能触发更新视图。

使用方式：`Vue.delete(target,propertyName)`，另外如果用`delete obj['property']`是不能更新页面的。

`Vue.set()`和`Vue.delete()`等同于两个实例方法：vm.$set()，vm.$delete()。`vm`表示的是`Vue`的实例。

所以也可以采用如下的方式：this.$set(c, "height", this.height);

③ `vm.$on`与`vm.$emit`

自定义组件实现双向绑定：

```
      <user-add @add-user="addUser" v-model="userInfo"></user-add>
```

等价以下的写法，也就是说`v-model`就是`v-bind`与`v-on`的语法糖。

```
<user-add
  v-bind:value="userInfo"
  v-on:input="userInfo = $event"
></user-add>
```

```
 Vue.component("user-add", {
        props: ["value"],
        template: `
            <div>
             <p>
                <input type="text" :value="value" @input="onInput" v-
on:keydown.enter="addUser" />
             </p>
             <button @click="addUser">新增用户</button>
              </div>
            `,
        methods: {
          addUser() {
            //将输入的用户数据通知给父组件，来完成新增用户操作.
            this.$emit("add-user");
          },
          onInput(e) {
            this.$emit("input", e.target.value);
          },
        },
      });
```

④使用插槽完成内容分发：在使用组件时，提供具体的数据内容，然后这些内容会插入到组件内部插槽的位置。

例如在使用弹窗组件的时候，不仅能传递窗口的内容，还能传递其它的内容，例如标题等。

```
<!-- 弹窗组件 -->
      <message :show="isShow" @close="closeWindow">
        <template v-slot:title>  <!-- titile的插槽 -->
          <h2>恭喜</h2>
        </template>
        <template>  <!-- 默认插槽 -->
          添加用户成功
        </template>
      </message>
```

```
//创建弹出的组件
      Vue.component("message", {
        //show表示的含义，控制弹出窗口的显示与隐藏。
        //slot:表示占坑。也就是窗口中的内容，是通过外部组件传递过来的。
        props: ["show"],
        template: `<div class='message-box' v-if="show">
             <!--具名插槽-->
             <slot name="title">默认标题</slot>
            <slot></slot>
            <span class="message-box-close" @click='$emit("close",false)'>关闭</span>
          </div>`,
      });
```

⑤ `vm.$on` 与 `vm.$emit` 应用：典型应用就是事件总线。

即通过在 `Vue` 原型上添加一个 `Vue` 实例作为事件总线，实现组件间相互通信，而且不受组件间关系的影响。在所有组件最上面创建事件总线，这样做的好处就是在任意组件中使用 `this.$bus` 访问到该 `Vue` 实例。

```
Vue.prototype.$bus=new Vue()
```

```
<message :show="showWarn" @close="closeWindow" class="warning">......
Vue.component("message", {
......
        mounted() {
          //给总线绑定`message-close`事件,也就是监听是否有`message-close`事件被触发。
          this.$bus.$on("message-close", () => {
            this.$emit("close", false);
          });
        },
      });
```

在窗口中添加一个“清空提示栏”按钮，单击该按钮的时候可以触发 `message-close` 事件，从而关闭提示窗口。

```
  <!-- 清空提示栏 -->
      <div class="toolbar">
        <button @click="$bus.$emit('message-close')">
          清空提示栏
        </button>
      </div>
```

⑥ `vm.$once` 与 `vm.$off`

`vm.$once` ：监听一个自定义事件，但是只触发一次。一旦触发之后，监听器就会被移除。

```
vm.$once('test', function (msg) { console.log(msg) })
```

`vm.$off`：移除自定义事件监听器。

```
vm.$off() // 如果没有提供参数，则移除所有的事件监听器；
vm.$off('test') // 如果只提供了事件，则移除该事件所有的监听器；
vm.$off('test', callback) // 如果同时提供了事件与回调，则只移除这个回调的监听器
```

⑦`ref`和`vm.$refs`

`ref`被用来给元素或子组件注册引用信息。`ref`是作为渲染结果被创建的，在初始渲染时不能访问它们，也就是必须在`mounted`构造函数中。`$refs`不是响应式的，不要试图用它在模板中做数据绑定。引用信息将会注册在父组件的`$refs`对象上，如果在普通的`DOM`元素上使用，引用指向的就是`DOM`元素；如果用在子组件上，引用就指向组件的实例。

```
<input type="text"  ref="inp" />
mounted(){  //mounted之后才能访问到inp
    this.$refs.inp.focus()
}
```

17、过滤器

在`Vue`中，过滤器的作用就是格式化数据，也就是对数据的过滤处理，比如将字符串格式化为首字母大写或者将日期格式化为指定的格式等。

```
Vue.filter('过滤器名称',function(value){ //自定义（全局）过滤器的语法 //value参数表示要处理的数据
  //过滤器业务逻辑，最终将处理后的数据进行返回
})
```

```
<div>{{msg|upper}}</div>
<div>{{msg|upper|lower}}</div>
```

而局部过滤器只能在其所定义的组件内使用。

```
        filters: {
          upper: function (value) {
            return value.charAt(0).toUpperCase() + value.slice(0);
          },
        },
```

而带参数的过滤器定义：

```
Vue.filter('format',function(value,arg1){ //value表示要过滤的数据 //arg1表示传递过来的参数
})
```

```
<div>
 {{data|format(`yyyy-MM-dd`)}}
</div>
```

18、自定义指令

基本的创建自定义指令语法：

```
Vue.directive('focus',{  //通过directive方法创建了一个focus指令。
    inserted:function(el){  //inserted表示的是指令的钩子函数，含义是：被绑定元素插入父节点时调用
     el.focus();       //获取元素焦点
    }
})
```

```
<input type="text" v-focus>  <!--自定义指令用法-->
```

带参数的自定义指令创建的语法：（改变元素背景色）

而`bind`这个钩子函数：只调用一次，第一次绑定指令到元素时调用，可以在此绑定只执行一次的初始化动作。

```
Vue.directive('color',{
    inserted:function(el,binding){    //binding表示传递过来的参数
        el.style.backgroundColor=binding.value.color;
    }
})
```

```
<input type="text" v-color='{color:"orange"}' />
```

自定义局部指令：局部指令的基本语法：局部指令只在所定义的组件中使用。

```
directives:{  //在Vue实例中添加directives
    focus:{        //指令的定义
        inserted:function(el){
            el.focus()
        }
    }
}
```

19、渲染函数

`Vue`推荐在绝大数情况下使用模板来创建`HTML`。然后在一些场景中，如果需要`JavaScript`的完全编程的能力，也就是使用`javaScript来创建HTML`，这时你可以用渲染函数，它比模板更接近编译器。

`Vue`通过建立一个虚拟DOM来追踪自己要如何改变真实`DOM`.

```
render:function(createElement){  //render函数的基本结构。
  retrun createElement(//createElement函数返回的结果为VNode,VNode是虚拟dom，用js对象模拟真实DOM
      tag, //标签名称 {string |Object|Function} 第一个参数，可以是字符串，也可以是对象或者是函数
       data,// 传递数据 第二个参数是对象，表示的是一个与模板中属性对应的数据对象。该参数可选
       children //第三个参数是一个数组,表示的是子节点数组
    )
}
```

```
 // heading组件
      //<heading :level="2" :title="title">{{title}}</heading> //这时要创建的组件
      // <h2 title="aaa"></h2> //这时上面的组件最终渲染的结果
      Vue.component("heading", {
        props: {
          level: {
            type: String,
            required: true,
          },
          title: {
```

```
            type: String,
            default: "",
          },
        },
        render(h) {
          return h(
            "h" + this.level, //参数1，表示要创建的元素
            { attrs: { title: this.title } }, //参数2（可选）
          this.$slots.default //参数3，子节点VNode数组，{{tile}}就是一个子元素
          );
        },
      });
```

20、函数式组件

组件没有管理任何状态，也没有监听任何传递给它的状态，也没有生命周期方法时，可以将组件标记为`functional`。这意味它无状态（没有响应式数据），也没有实例（没有`this`上下文）。因为只是函数，所以渲染的开销相对较小。函数化的组件中的 Render 函数，提供了第二个参数 context 作为上下文，data、props、slots、children 以及 parent 都可以通过 context 来访问。

21、混入

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被"混合"进入该组件本身的选项。

```
// 定义一个混入对象
var myMixin={created:function(){......},methods:{......}}
Vue.component('comp',{
    mixins:[myMixin]
})
```

22、插件

混入，组件封装等都可提高组件的复用功能，但这种方式不适合分发，也就是不适合将这些内容上传到`github`上，`npm`上。而这种情况最适合通过插件实现。插件通常用来为`Vue`添加全局功能。插件的功能范围一般有：

①添加全局方法或属性，如'element' ②添加全局资源 ③通过全局混入来添加一些组件选项，如`vue-router`

④添加`vue实例`方法，通过把它们添加到`Vue.prototype`上实现

⑤一个库，提供自己的`API`，同时提供上面提到的一个或多个功能，例如`vue-router`

插件声明：`Vue.js`的插件应该暴露一个`install`方法。这个方法的第一个参数是`Vue`构造器，第二个参数是一个可选的选项对象。

```
MyPlugin.install = function (Vue, options) {
  // 1. 添加全局方法或 property
  Vue.myGlobalMethod = function () {
    // 逻辑...
  }
  // 2. 添加全局资源
  Vue.directive('my-directive', {
    bind (el, binding, vnode, oldVnode) {
      // 逻辑...
    }
    ...
  })
  // 3. 注入组件选项
```

```
  Vue.mixin({
    created: function () {
      // 逻辑...
    }
    ...
  })
  // 4. 添加实例方法
  Vue.prototype.$myMethod = function (methodOptions) {
    // 逻辑...
  }
}
```