第 3 章

Vue.js 基础

3.1 MVVM 模式

MVVM(Model-View-ViewModel,模型-视图-视图模型)是一种软件架构模式,用于将用户界面与业务逻辑分离,中间由试图模型作为桥梁连接。

1) Model(模型)

Model 代表应用程序的数据和业务逻辑。它可以是从服务器获取的数据、本地存储的数据或其他数据源。Model 通常以类或对象的形式表示。

2) View(视图)

View 是用户界面的可见部分,即用户可以看到和与之交互的部分。View 通常是由 HTML、CSS 和 UI 控件组成,负责将 Model 中的数据呈现给用户,并将用户的输入反馈给 ViewModel。

3) ViewModel(视图模型)

ViewModel 是连接 View 和 Model 之间的桥梁,负责管理视图的数据和行为,并提供与视图交互所需的方法和命令。ViewModel 的主要目的是将业务逻辑和视图分离,使得视图可以专注于展示数据和用户交互,而不需要关注具体的数据获取和处理细节。

MVVM 模式的核心特性是双向数据绑定。当模型中的数据改变时,视图会自动更新以反映最新的数据;反之,当用户在视图中输入或修改数据时,模型也会相应地更新。

Vue.js 采用 MVVM 模式,当用户修改数据时,视图能够自动更新。同样,当用户修改视图时,数据也会相应地更新。这个过程完全由 Vue.js 处理,不需要额外的操作。这种双向绑定的特性使得开发者能够更专注于业务逻辑,而不必过多关注数据与视图之间的同步问题。

3.2 数据绑定与插值

在 Vue.js 中,数据绑定和插值是实现动态数据展示的重要机制。数据绑定允许你将 Vue.js 实例中的数据与 DOM 元素进行关联,以实现数据的自动更新。这意味着当 Vue.js

实例中的数据发生变化时，与之绑定的 DOM 元素也会自动更新，从而保持视图和数据的同步。插值是一种在模板中嵌入动态数据的方式。通过使用双大括号“{{ }}”将 Vue.js 实例的数据绑定到模板中，可以在视图中展示实时的数据内容。

3.2.1 文本绑定

【例 3-1】 使用 Mustache 语法（双大括号语法）进行文本插值。修改 App.vue，代码如下所示。

```
<script>
export default {
  data() {
    return {
      msg: "hello vue",
    };
  },
};
</script>

<template>
  <div>
    {{ msg }}
  </div>
</template>
```

使用 VS Code 快捷键 Ctrl＋S 保存页面代码之后，可以看到之前用 Vite 启动的 Vue.js 程序页面，页面显示内容更新为 hello vue，如图 3-1 所示。

图 3-1 更新为 hello vue

<script>标签中使用 data 方法返回一个对象，该对象包含一个名为 msg 的数据属性，其初始值为 hello vue。data 方法是 Vue.js 组件中用于定义响应式数据的地方。通过返回的对象中的属性，我们可以在模板中直接访问和展示这些数据。

<template>标签定义了组件的模板结构。{{ msg }}是 Vue.js 中的插值语法，用于将

组件的数据动态插入模板中。在这里，msg 数据属性的值会被渲染到< div >元素中。

这个组件的作用是显示一个包含 hello vue 文本的< div >元素。当组件的 msg 数据属性发生变化时，对应的模板会自动更新以反映新的值。例如，将 msg："hello vue"修改为 msg："hello"，保存之后，页面显示内容更新为 hello。

3.2.2 HTML 代码绑定

【例 3-2】 双大括号会将数据解析为普通文本，而非 HTML 代码。修改 App.vue，代码如下所示。

```
<script>
export default {
  data() {
    return {
      msg: "<h1>hello vue</h1>",
    };
  },
};
</script>

<template>
  <div>
    {{ msg }}
  </div>
</template>
```

保存代码之后，页面显示为< h1 > hello vue </h1 >。这里的 h1 只作为字符串输出。

【例 3-3】 为了输出真正的 HTML 代码，需要使用 v-html 指令。修改 App.vue，代码如下所示。

```
<script>
export default {
  data() {
    return {
      msg: "<h1>hello vue</h1>",
    };
  },
};
</script>

<template>
  <div v-html="msg"></div>
</template>
```

保存代码之后，页面显示为大字体的 hello vue。以 Windows 为例，在浏览器页面按 F12 键打开浏览器控制面板，单击 Elements 选项，查看 DOM 结构，可以看到 h1 标签，如图 3-2 所示。

图 3-2　查看 DOM 结构

3.2.3　属性绑定

属性绑定是在 Vue.js 中用于将数据动态地绑定到 HTML 元素的属性上。通过属性绑定，我们可以实现根据数据的变化，自动更新 HTML 元素的属性值。

【例 3-4】　使用 v-bind 指令实现属性绑定，修改 App.vue，代码如下所示。

```
<script>
export default {
  data() {
    return {
      url: "http://www.tup.tsinghua.edu.cn/index.html",
    };
  },
};
</script>

<template>
  <a v-bind:href = "url" target = "_blank">清华大学出版社</a>
</template>
```

保存代码之后，页面显示为清华大学出版社，单击“清华大学出版社”可跳转至清华大学出版社官网。这里通过 v-bind 指令给 href 属性动态赋值，将在 data 中定义的 url 绑定到 href 属性上。除了 href 属性，如 class、style 属性等，也可以使用 v-bind 指令进行属性绑定，从而动态赋值。

v-bind 指令可以简写为冒号(:)形式。例如，上面 template 中代码可改为：

```
<template>
  <a :href="url" target="_blank">清华大学出版社</a>
</template>
```

v-bind:href 等价于：href。

3.2.4 JavaScript 表达式绑定

JavaScript 表达式绑定是在 Vue.js 中用于将 JavaScript 表达式的结果动态地绑定到 HTML 元素上的一种方式。

【例 3-5】 修改 App.vue，代码如下所示。

```
<script>
export default {
  data() {
    return {
      flag: true,
    };
  },
};
</script>

<template>
  <div>
    <span>{{ flag ? "true" : "false" }}</span>
  </div>
</template>
```

在中使用插值表达式{{ flag ? "true" : "false" }}来动态展示 flag 数据属性的值。如果 flag 为 true，则显示"true"，否则显示"false"。

通过上面的例子可知，双大括号语法不仅可以绑定单一键值，还可以绑定 JavaScript 表达式。但要注意的是，不能直接在{{ }}中使用 JavaScript 的条件语句(如 if 语句)。对于条件语句，可以使用 v-if、v-else-if 和 v-else 等指令执行条件渲染。

3.3 方法选项

上述在 export default 中定义的 data 函数称为数据选项，定义的属性能够在 template 中直接访问。接下来介绍方法选项 methods，用来改变 data 函数中定义的属性值。

【例 3-6】 修改 App.vue，代码如下所示。

```
<script>
export default {
  data() {
    return {
      count: 0,
    };
```

```
    },
    methods: {
      increment() {
        this.count++;
      },
    },
  };
  </script>

  <template>
    <div>
      <button v-on:click="increment">Increment</button>
      <p>{{ count }}</p>
    </div>
  </template>
```

在上述示例中，在 methods 选项中定义了一个名为 increment 的方法。在按钮上使用 v-on：click 绑定单击事件，当单击按钮时，increment 方法会被调用，然后它会将 count 数据属性增加 1。通过这种方式，实现了单击按钮后计数器值的增加。

需要注意的是，在方法选项 methods 中不能使用箭头函数。使用箭头函数会导致 this 不会指向组件实例，进而无法访问组件实例的属性和数据。为了确保正确的上下文绑定，在方法选项 methods 中，应该使用普通的函数语法来定义方法。

通过使用 methods 选项，可以在组件中定义可重用的方法，并在需要的地方调用和使用这些方法，使组件具有交互性。

v-on：可以简写为“@”。例如，上面 template 中代码可改为：

```
  <template>
    <div>
      <button @click="increment">Increment</button>
      <p>{{ count }}</p>
    </div>
  </template>
```

v-on：click 等价于@click。

【例 3-7】 再实现一个数据绑定与插值的综合应用示例。修改 App.vue，代码如下所示。

```
  <script>
  export default {
    data() {
      return {
        title: "数据绑定与插值综合应用示例",
        message: "初始消息",
        isDisabled: false,
      };
    },
    methods: {
```

```
    updateMessage() {
      this.message = "更新后的消息";
      this.isDisabled = true;
    },
  },
};
</script>

<template>
  <div>
    <h2>{{ title }}</h2>
    <p>{{ message }}</p>
    <button v-bind:disabled="isDisabled" v-on:click="updateMessage">
      更新消息
    </button>
  </div>
</template>
```

在<script>部分，在 data 选项中定义了三个数据属性：title、message 和 isDisabled，在 methods 选项中定义了一个方法 updateMessage，用于更新 message 的值并将 isDisabled 设置为 true。

在<template>部分，使用双大括号插值语法将 title 和 message 数据绑定到<h2>和<p>元素中。使用 v-bind:disabled 指令将 isDisabled 数据绑定到按钮的 disabled 属性，控制按钮的可单击状态。通过 v-on:click="updateMessage"将 updateMessage 方法绑定到按钮的单击事件。

单击“更新消息”按钮，按钮名称变为“更新后的消息”，并且按钮变为不可单击状态。

3.4　选项式 API 生命周期

Vue.js 提供了一系列的生命周期钩子函数，允许开发者在不同的阶段添加代码来执行特定的逻辑。下面是选项式 API 生命周期钩子函数的介绍。

1）beforeCreate

用于在组件实例被创建之前执行一些初始化逻辑。在这个阶段，可以进行一些组件实例的设置，例如添加全局事件监听器、初始化非响应式数据等。

```
beforeCreate() {
  console.log('beforeCreate');
}
```

2）created

在组件实例创建完成后调用。在这个阶段，组件实例已经创建，可以访问和操作组件的数据、方法以及其他选项，常用来请求接口。

```
created() {
  console.log('created');
}
```

3）beforeMount

在组件挂载到 DOM 之前调用。在这个阶段，组件的模板已经编译完成，但尚未渲染到真实的 DOM 结构中。在这个钩子函数中，可以执行一些在挂载之前需要准备的操作，例如修改数据、计算属性等。

```
beforeMount() {
  console.log('beforeMount');
}
```

4）mounted

在组件挂载到 DOM 之后调用。在这个阶段，组件的模板已经渲染到 DOM 中，并且可以进行 DOM 操作。在这个钩子函数中，可以执行一些需要依赖 DOM 的操作，例如初始化第三方库、添加 DOM 事件监听器等。

```
mounted() {
  console.log('mounted');
}
```

5）beforeUpdate

在组件更新之前调用。在这个阶段，组件的数据发生变化，但 DOM 尚未更新。在这个钩子函数中，可以执行一些在更新之前需要准备的操作，例如保存一些临时数据。

```
beforeUpdate() {
  console.log('beforeUpdate');
}
```

6）updated

在组件更新完成后调用。在这个阶段，组件的数据已经更新到最新值，并且 DOM 已经更新。在这个钩子函数中，可以执行一些在更新后需要进行的操作，例如获取更新后的 DOM 节点、重新计算数据等。

```
updated() {
  console.log('updated');
}
```

7）beforeUnmount

在组件卸载之前调用。在这个阶段，组件尚未从 DOM 中移除。在这个钩子函数中，可以执行一些在卸载之前需要清理的操作，例如取消定时器、清除事件监听器等。

```
beforeUnmount() {
  console.log('beforeUnmount');
}
```

8）unmounted

在组件卸载之后调用。在这个阶段，组件已经从 DOM 中移除，并且组件实例将被销毁。在这个钩子函数中，可以执行一些在卸载之后的清理操作，例如释放内存、清除引用等。

```
unmounted() {
  console.log('unmounted');
}
```

通过使用生命周期钩子函数，你可以在组件的不同阶段执行适当的代码，以满足特定的需求。上述选项式 API 生命周期由执行先后顺序进行排列。

【例 3-8】 修改子组件 HelloWorld.vue，代码如下所示。

```
<script>
export default {
  data() {
    return {
      message: "Hello Vue!",
    };
  },
  beforeCreate() {
    console.log("beforeCreate");
  },
  created() {
    console.log("created");
  },
  beforeMount() {
    console.log("beforeMount");
  },
  mounted() {
    console.log("mounted");
  },
  beforeUpdate() {
    console.log("beforeUpdate");
  },
  updated() {
    console.log("updated");
  },
  beforeUnmount() {
    console.log("beforeUnmount");
  },
  unmounted() {
    console.log("unmounted");
  },
  methods: {
    updateMessage() {
      this.message = "Updated Message";
    },
  },
```

```
};
</script>

<template>
  <div>
    <h2>{{ message }}</h2>
    <button @click="updateMessage">Update Message</button>
  </div>
</template>
```

【例 3-9】 修改父组件 App.vue，代码如下所示。

```
<script>
import HelloWorld from "./components/HelloWorld.vue";

export default {
  components: {
    HelloWorld,
  },
  data() {
    return {
      status: true,
    };
  },
  methods: {
    unmountComponent() {
      this.status = false;
    },
  },
};
</script>

<template>
  <div>
    <div v-if="status"><HelloWorld /></div>
    <div v-else>组件已卸载</div>
    <button @click="unmountComponent">卸载组件</button>
  </div>
</template>
```

保存之后，以 Windows 为例，在浏览器页面按 F12 键，打开浏览器控制面板，单击 Console 选项，查看打印信息，可以看到生命周期执行顺序，如图 3-3 所示。

当组件加载时，生命周期执行顺序为 beforeCreate→created→beforeMount→mounted。

当单击 Update Message 按钮时，updateMessage 方法会被调用，使得数据更新。在数据更新时，生命周期执行顺序为 beforeUpdate→updated。

当单击“卸载组件”按钮时，子组件会被卸载，生命周期执行顺序为 beforeUnmount→unmounted。

图 3-3　生命周期执行顺序

通过观察控制台输出，可以更好地理解每个钩子函数在组件生命周期中的执行顺序。总的来说，组件的生命周期执行顺序为 beforeCreate→created→beforeMount→mounted→beforeUpdate→updated→beforeUnmount→unmounted。

除了上述生命周期，还有 setup 以及 keep-alive 组件的 activated 与 deactivated。setup 将在 4.5 节中详细介绍，activated 与 deactivated 将在 5.4 节中详细介绍。

3.5　基本指令

指令是 Vue.js 中的一种特殊语法，用于在模板中添加特定的行为和功能。指令以 v-开头，并通过绑定到 DOM 元素上来实现相应的功能。

3.5.1　v-text

v-text 指令用于将数据动态地设置为元素的纯文本内容。当使用 v-text 指令时，元素的内容将被替换为指定数据的值，如字符串、变量、表达式等。该指令会将数据的值作为纯文本进行渲染，不会将内容解释为 HTML。

【例 3-10】　修改 App.vue，代码如下所示。

```
<script>
export default {
  data() {
    return {
      message: "Hello Vue!",
    };
  },
};
</script>
```

```
<template>
  <div v-text="message"></div>
</template>
```

在上述示例中，message 是 Vue 实例中定义的一个属性。通过使用 v-text 指令，元素的内容将被替换为 message 的值。无论 message 是字符串(string)类型、数字(number)类型还是其他类型，都会以纯文本的形式显示在 div 元素中。

与双大括号语法相比，v-text 指令提供了一种更显式地将数据作为纯文本进行渲染的方式。

3.5.2 v-html

v-html 指令用于将数据动态地设置为 HTML 内容。

【例 3-11】 修改 App.vue，代码如下所示。

```
<script>
export default {
  data() {
    return {
      msg: "<h1>hello vue</h1>",
    };
  },
};
</script>

<template>
  <div v-html="msg"></div>
</template>
```

需要注意的是，使用 v-html 指令时要谨慎，确保所绑定的 HTML 内容是可信的，以避免潜在的安全风险。Vue.js 会将绑定的 HTML 内容直接渲染到页面上，如果内容来源不可信，可能会导致 XSS 攻击。

拓展： XSS 攻击。

XSS(Cross-Site Scripting，跨站脚本)攻击是一种常见的网络安全漏洞，它允许攻击者将恶意脚本注入受信任的网站中，然后在用户的浏览器中执行这些恶意脚本。Cross-Site Scripting 之所以称为 XSS，是为了避免与 CSS(层叠样式表)混淆。这种攻击利用了网页应用程序对用户输入数据的不当处理，导致恶意脚本被插入网页中，并被用户的浏览器执行，从而导致恶意行为的发生。

3.5.3 v-bind

v-bind 指令用于将数据绑定到 HTML 元素的属性上，使其可以动态地根据数据的变化来更新属性的值。

【例 3-12】 修改 App.vue，代码如下所示。

```
<script>
export default {
  data() {
    return {
      url: "http://www.tup.tsinghua.edu.cn/index.html",
    };
  },
};
</script>

<template>
  <a v-bind:href="url" target="_blank">清华大学出版社</a>
</template>
```

url 是 data 选项中的一个属性，用于存储链接的 URL 地址。通过使用 v-bind 指令，将 url 与 href 属性进行绑定，使得<a>元素的 href 属性的值始终与 url 的值保持一致。

v-bind 指令可以简写为冒号(:)形式，如下所示。

```
<template>
  <a :href="url" target="_blank">清华大学出版社</a>
</template>
```

3.5.4 v-on

v-on 指令用于在模板中监听 DOM 事件，并在事件触发时执行指定的方法。

【例 3-13】 修改 App.vue，代码如下所示。

```
<script>
export default {
  data() {
    return {
      count: 0,
    };
  },
  methods: {
    increment() {
      this.count++;
    },
  },
};
</script>

<template>
  <div>
    <button v-on:click="increment">Increment</button>
    <p>{{ count }}</p>
  </div>
</template>
```

在上述示例中，v-on:click 指令将绑定到< button >元素上，并监听单击事件。当按钮被单击时，将执行 increment 方法。

v-on 指令可以简写为@，如下所示。

```
< template >
  < div >
    < button @click = "increment"> Increment </button >
    < p >{{ count }}</p >
  </div >
</template >
```

除了 click 事件，还可以使用 v-on 指令来监听其他常见的 DOM 事件，如 input、submit 和 keydown 等。

【例 3-14】 再以 input 事件为例，修改 App. vue，代码如下所示。

```
< script >
export default {
  methods: {
    handleInput(event) {
      //处理输入事件的逻辑
      console.log(event.target.value);
    },
  },
};
</script >

< template >
  < div >
    < input @input = "handleInput" />
  </div >
</template >
```

在 v-on 指令中，还可以给绑定的方法传递参数。例如，可以将事件对象 $event 作为参数传递给方法，以便在方法中访问事件的相关信息。

【例 3-15】 修改 App. vue，代码如下所示。

```
< script >
export default {
  methods: {
    handleClick(e) {
      console.log(e);
    },
  },
};
</script >

< template >
  < div >
    < button @click = "handleClick( $event)"> Increment </button >
```

```
  </div>
</template>
```

保存代码之后，刷新浏览器页面。以 Windows 为例，在浏览器页面按 F12 键，打开浏览器面板，单击 Console 选项，单击 Increment 按钮，查看打印信息，如图 3-4 所示。

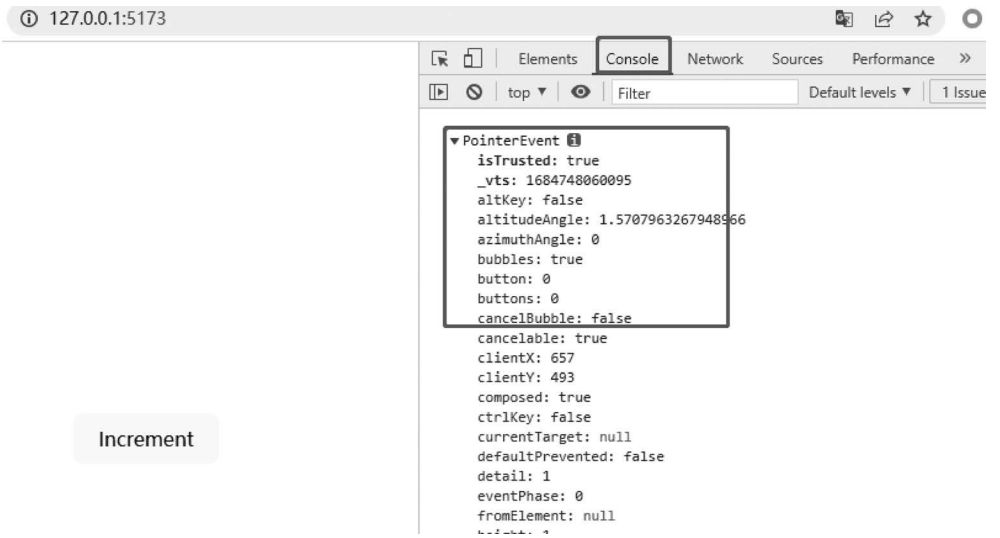

图 3-4 单击 Increment 按钮，查看打印信息

3.5.5 v-show

v-show 指令用于根据条件动态地控制元素的显示与隐藏。使用 v-show 指令时，元素的显示与隐藏状态是通过 CSS 的 display 属性进行控制的。当指令绑定的表达式的值为真(true)时，元素将显示出来；当表达式的值为假(false)时，元素将被隐藏起来。

【例 3-16】 修改 App.vue，代码如下所示。

```
<script>
export default {
  data() {
    return {
      isVisible: true,
    };
  },
  methods: {
    toggleVisibility() {
      this.isVisible = !this.isVisible;
    },
  },
};
</script>
```

```
<template>
  <div>
    <button @click="toggleVisibility">切换显示</button>
    <div v-show="isVisible">这是可见的内容</div>
  </div>
</template>
```

在上述示例中，有一个按钮和一个 元素，通过单击按钮可以切换 元素的显示与隐藏状态。初始状态下，isVisible 的值为 true，因此 元素会被显示出来。当单击按钮时，会触发 toggleVisibility 方法，该方法会将 isVisible 的值取反，div 元素将通过设置 display：none 被隐藏起来。

使用 v-show 指令的好处是，元素隐藏时仍然存在于 DOM 中，仅通过修改 display 样式便可实现隐藏效果。这样可以避免频繁地创建和销毁元素，适用于需要频繁切换显示状态的场景。

3.5.6 v-if

1. v-if 概述

v-if 指令用于根据条件动态地添加或移除元素，实现条件渲染。使用 v-if 指令时，元素的渲染与移除是根据绑定的表达式的值来决定的。如果表达式的值为真(true)，则元素会被创建并插入 DOM 中；如果表达式的值为假(false)，则元素会从 DOM 中移除。

【例 3-17】 修改 App.vue，代码如下所示。

```
<script>
export default {
  data() {
    return {
      isVisible: true
    };
  },
  methods: {
    toggleVisibility() {
      this.isVisible = !this.isVisible;
    }
  }
};
</script>

<template>
  <div>
    <button @click="toggleVisibility">切换显示</button>
    <div v-if="isVisible">这是可见的内容</div>
  </div>
</template>
```

初始状态下，isVisible 的值为 true，因此 元素会被创建并插入 DOM 中。当单击

按钮时，触发 toggleVisibility 方法，该方法会将 isVisible 的值取反，< div >元素会从 DOM 中移除。

当使用 v-if 指令时：若条件为假，则元素会被完全从 DOM 中移除，不再占据空间，事件监听器也会被销毁；若条件为真，则元素会被重新创建并插入 DOM 中。

需要注意的是，由于 v-if 指令涉及 DOM 的创建和销毁，频繁地切换元素的显示与隐藏会有一定的性能开销。因此，v-if 适用于条件变化不频繁的场景。

2. v-show 与 v-if 区别

v-show 和 v-if 是 Vue 中用于条件渲染的两种指令，它们在使用方式和行为上存在以下区别：

(1) 显示与隐藏方式：v-show 通过修改元素的 CSS 属性来控制其显示与隐藏，而 v-if 是根据条件动态地添加或移除元素。

(2) 初始化渲染开销：v-show 会始终渲染元素，仅通过样式控制显示与隐藏。因此，在初始渲染时会产生一定的开销。相比之下，v-if 在初始渲染时，如果条件为假，则不会渲染对应元素；只有在条件为真时才会进行渲染。

(3) 切换开销：v-show 切换元素的显示与隐藏仅涉及修改元素的 display 样式，不涉及元素的创建或销毁，因此切换开销较低。而 v-if 在条件发生变化时，会根据条件的真假来动态地添加或移除元素，由于涉及元素的创建和销毁，因此切换开销较高。

综上所述，v-show 适用于需要频繁切换显示与隐藏的场景，因为它不涉及元素的创建和销毁。而 v-if 更适合条件变化较少，且切换显示与隐藏时需要重新销毁和重建组件的场景。

3.5.7 v-else

v-else 指令用于在 v-if 指令的条件不满足时，渲染一个备用的元素或组件。v-else 指令必须紧跟在带有 v-if 指令的元素后面，并且没有任何额外的条件。它会在前面的 v-if 指令的条件不满足时，自动渲染。

【例 3-18】 使用 v-if 和 v-else 实现一个简单的登录和注销功能。修改 App.vue，代码如下所示。

```
<script>
export default {
  data() {

      isLoggedIn: false,
      username: "",
    };
  },
  methods: {
    login() {
      //模拟登录
      this.isLoggedIn = true;
```

```
        this.username = "qinghua";
      },
      logout() {
        //模拟退出登录
        this.isLoggedIn = false;
        this.username = "";
      },
    },
  };
  </script>

  <template>
    <div>
      <h2 v-if="isLoggedIn">欢迎回来,{{ username }}!</h2>
      <p v-else>请先登录。</p>

      <button @click="login">登录</button>
      <button @click="logout">退出登录</button>
    </div>
  </template>
```

在 data 选项中，有两个属性：isLoggedIn 和 username。isLoggedIn 用于跟踪用户是否已登录，初始值为 false(表示未登录)；username 用于存储登录用户的用户名，初始值为空字符串。

在 methods 选项中，有两个方法：login 和 logout。当 login 方法被调用时，将 isLoggedIn 设置为 true，表示用户已登录，并将 username 设置为 qinghua。当 logout 方法被调用时，将 isLoggedIn 设置为 false，表示用户已注销，并清空 username。

在模板中，根据 isLoggedIn 的值进行条件渲染。如果 isLoggedIn 的值为 true，则显示欢迎消息，包含用户的用户名；如果 isLoggedIn 的值为 false，则显示请登录的提示消息。模板还包括两个按钮，分别绑定了 login 和 logout 方法，单击按钮会触发相应的方法。

3.5.8 v-else-if

v-else-if 指令用于在多个条件之间进行选择，并在满足条件时渲染相应的内容。

【例 3-19】 修改 App.vue，代码如下所示。

```
<script>
export default {
  data() {
    return {
      status: "success",
    };
  },
};
</script>
```

```
<template>
  <div>
    <p v-if="status === 'pending'">请求正在进行中...</p>
    <p v-else-if="status === 'success'">请求成功!</p>
    <p v-else-if="status === 'error'">请求失败,请重试。</p>
    <p v-else>未知状态。</p>
  </div>
</template>
```

使用 v-if 指令来判断 status 的值是否为 pending。如果 status 的值是 pending,则渲染显示“请求正在进行中...”的段落;如果 status 的值不是 pending,则使用 v-else-if 指令判断 status 的值是否为 success。如果 status 的值是 success,则渲染显示“请求成功!”的段落。接着,使用 v-else-if 指令判断 status 的值是否为 error,如果是,则渲染显示“请求失败,请重试。”的段落。最后,如果 status 的值均不是 pending、success、error 中的任何一个,那么使用 v-else 指令渲染显示“未知状态。”的段落。

在上述示例中,由于 status 的值为 success,所以将渲染一条“请求成功”的消息。可以根据需要修改 status 的值,以观察不同条件下的渲染结果。

3.5.9 v-for

v-for 指令用于循环渲染数据。它可以在模板中遍历数组或对象,并为每个元素或属性生成对应的 DOM 元素或组件实例。

1. 循环数组

【例 3-20】 修改 App.vue,代码如下所示。

```
<script>
export default {
  data() {
    return {
      items: [{ name: "Apple" }, { name: "Banana" }, { name: "Orange" }],
    };
  },
};
</script>

<template>
  <div>
    <ul>
      <li v-for="(item, index) in items" :key="index">
        {{ item.name }}
      </li>
    </ul>
  </div>
</template>
```

在模板部分，使用 v-for 指令遍历 items 数组，并为数组中的每个元素创建一个< li >标签。在 v-for 中，使用括号来同时访问数组元素和对应的索引。其中，item 表示数组中的元素对象，index 表示数组中元素的索引。将索引通过：key 绑定，为每个< li >标签提供一个唯一的 key。

在每个< li >标签中，使用插值语法{{ item. name }}显示每个对象的 name 属性的值，这样会在页面上渲染出一个无序列表(< ul >)，其包含三个列表项(< li >)，每个列表项显示了数组中的一个对象的 name 属性。在本示例中，页面将被渲染为 Apple、Banana 和 Orange。

2. 循环对象

【例 3-21】 修改 App. vue，代码如下所示。

```
<script>
export default {
  data() {
    return {
      object: {
        name: "qinghua",
        age: 25,
        email: "qinghua@example.com",
      },
    };
  },
};
</script>

<template>
  <div>
    <ul>
      <li v-for="(value, key) in object" :key="key">{{ key }}: {{ value }}</li>
    </ul>
  </div>
</template>
```

在模板部分，使用 v-for 指令遍历 object 对象，并为对象中的每个键值对创建一个< li >标签。在 v-for 中，使用括号来同时访问键值对的值和对应的键名。其中，value 表示键值对的值，key 表示键值对的键名。

3.5.10 v-model

v-model 指令用于在表单输入和数据之间提供双向数据绑定。它功能强大，简化了用户输入和组件状态之间的同步。

v-model 指令可用于各种表单元素，如< input >、< textarea >和< select >。下面分别展示 v-model 在输入框、复选框和下拉选择框中的使用。

1）输入框

【例 3-22】 修改 App.vue，代码如下所示。

```
<script>
export default {
  data() {
    return {
      message: "",
    };
  },
};
</script>

<template>
  <div>
    <input v-model="message" type="text" />
    <p>输入框的值为：{{ message }}</p>
  </div>
</template>
```

2）复选框

【例 3-23】 查看完输入框效果后，重新修改 App.vue，代码如下所示。

```
<script>
export default {
  data() {
    return {
      isChecked: false,
    };
  },
};
</script>

<template>
  <div>
    <input v-model="isChecked" type="checkbox" />
    <label>复选框</label>
    <p>复选框是否选中：{{ isChecked }}</p>
  </div>
</template>
```

3）下拉选择框

【例 3-24】 查看完复选框效果后，重新修改 App.vue，代码如下所示。

```
<script>
export default {
  data() {
    return {
      selectedOption: "",
    };
```

```
  },
};
</script>

<template>
  <div>
    <select v-model="selectedOption">
      <option value="option1">选项 1</option>
      <option value="option2">选项 2</option>
      <option value="option3">选项 3</option>
    </select>
    <p>选中的选项：{{ selectedOption }}</p>
  </div>
</template>
```

在每个示例中，v-model 指令都将表单元素的值绑定到相应的数据属性上（在 data 选项中定义的属性）。对表单元素的任何更改都会自动更新关联的数据属性，反之亦然。

除了这些常用的基本指令，还有 v-slot、v-pre、v-once 和 v-memo 等高级指令。高级指令将在 Vue.js 进阶篇中介绍。

3.6 计算属性选项

计算属性选项（computed）允许你基于现有的数据属性进行计算，并返回一个新的属性。这些计算属性会根据其所依赖的响应式数据进行缓存，只有当依赖的数据发生变化时才会重新计算。

【例 3-25】 修改 App.vue，代码如下所示。

```
<script>
export default {
  data() {
    return {
      price: 100,
      discount: 0.2,
    };
  },
  computed: {
    discountedPrice() {
      return this.price - this.price * this.discount;
    },
  },
};
</script>

<template>
  <div>
    <p>原始价格：{{ price }}</p>
```

```
    <p>折扣后的价格: {{ discountedPrice }}</p>
  </div>
</template>
```

在这个示例中,price 表示商品的原价格,discount 表示折扣比例。在 computed 选项中定义一个计算属性 discountedPrice 表示计算折扣后的价格。只有当依赖的 price 或 discount 发生变化,计算属性 discountedPrice 才会自动重新计算,否则直接返回缓存中的值。

使用计算属性可以提高代码的可读性和维护性,尤其是当你需要根据多个数据属性进行复杂的计算时。计算属性还能够利用缓存机制提高性能,避免不必要的重复计算。

3.7 监听器选项

监听器选项(watch)用于监视数据的变化并执行相应的操作。通过 watch 选项,我们可以监听指定的数据属性,当这些属性发生变化时,会触发相应的回调函数,从而执行自定义的逻辑,例如发送请求、更新其他数据和触发方法等。

3.7.1 默认懒执行

【例 3-26】 修改 App.vue,代码如下所示。

```
<script>
export default {
  data() {
    return {
      count: 0,
    };
  },
  watch: {
    count(newValue, oldValue) {
      console.log(`计数从 ${oldValue} 变为 ${newValue}`);
    },
  },
  methods: {
    increment() {
      this.count++;
    },
  },
};
</script>

<template>
  <div>
    <p>当前计数: {{ count }}</p>
    <button @click="increment">增加</button>
  </div>
</template>
```

在这个示例中，使用 watch 选项来监视 count 属性的变化。当 count 发生变化时，watch 选项中指定的回调函数会被调用。在回调函数中，可以访问两个参数：newValue 和 oldValue，它们分别表示属性的新值和旧值。

在模板中，使用@click 指令绑定了一个单击事件来增加计数值。每次单击按钮时，count 的值会增加，并触发 watch 选项中的回调函数，从而在浏览器控制台中输出打印的值。

值得注意的是，在浏览器控制台中，刚开始是没有打印信息的，只有单击“增加”按钮时，控制台才打印信息，说明 watch 选项默认是懒执行的，即第一次不执行，如图 3-5 和图 3-6 所示。

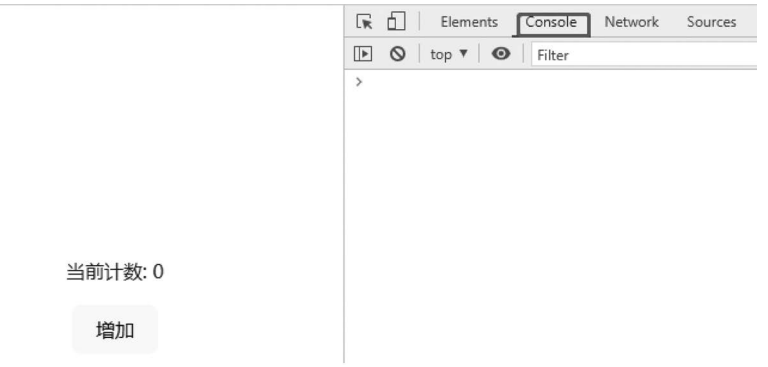

图 3-5　第一次 watch 不执行

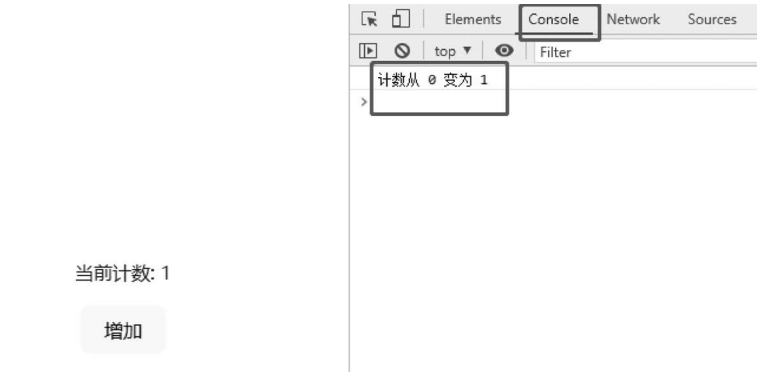

图 3-6　单击增加按钮后，watch 执行

3.7.2　立即执行

当需要在组件初始化时立即对某个数据进行处理或执行一些初始化逻辑时，可以使用 immediate：true。这样可以确保在组件挂载后立即执行相应的操作，而不需要等待数据的变化。

【例 3-27】　修改 App.vue，代码如下所示。

```
<script>
export default {
  data() {
    return {
      count: 0,
    };
  },
  watch: {
    count: {                //count 从函数变为对象
      immediate: true,    //新增 immediate
      handler(newValue, oldValue) {
        console.log(`计数从 ${oldValue} 变为 ${newValue}`);
      },
    },
  },
  methods: {
    increment() {
      this.count++;
    },
  },
};
</script>

<template>
  <div>
    <p>当前计数：{{ count }}</p>
    <button @click="increment">增加</button>
  </div>
</template>
```

在浏览器控制台中，可以看到打印信息“计数从 undefined 变为 0”，说明 watch 监听器立即执行了，如图 3-7 所示。

图 3-7　watch 监听器立即执行

3.7.3 深度监听

watch 监听默认是浅层的，无法监听嵌套属性值的变动。

【例 3-28】 修改 App.vue，代码如下所示。

```
<script>
export default {
  data() {
    return {
      person:{
        name:'小李',
        age:20
      }
    };
  },
  watch: {
    count: {
      person(newValue, oldValue) {
        console.log(`计数从 ${oldValue} 变为 ${newValue}`);
      },
    },
  },
  methods: {
    increment() {
      this.person.age++;
    },
  },
};
</script>

<template>
  <div>
    <p>姓名：{{ person.name }} -- 年龄：{{ person.age }}</p>
    <button @click="increment">年龄增加 1</button>
  </div>
</template>
```

保存之后，单击页面“年龄增加 1”按钮，会发现无论单击多少次，控制台都没有打印信息。可以使用 deep:true 来进行深度监听。

【例 3-29】 修改 App.vue，代码如下所示。

```
<script>
export default {
  data() {
    return {
      person:{
        name:'小李',
        age:20
      }
```

```
      };
    },
    watch: {
      person: {
        deep:true,//新增 deep
        handler(newValue, oldValue) {
          console.log(`计数从 ${oldValue} 变为 ${newValue}`);
        },
      },
    },
    methods: {
      increment() {
        this.person.age++;
      },
    },
};
</script>

<template>
  <div>
    <p>姓名: {{ person.name }} -- 年龄: {{ person.age }}</p>
    <button @click="increment">年龄增加 1</button>
  </div>
</template>
```

保存之后，单击页面“年龄增加 1”按钮，会发现浏览器控制台打印出信息“计数从 [object Object] 变为 [object Object]”。

值得注意的是，这里提到的是 watch 无法监听嵌套属性值的变动，如果直接给整个对象赋值，watch 是可以监听到的。

【例 3-30】 修改 App.vue，代码如下所示。

```
<script>
export default {
  data() {
    return {
      person: {
        name: '小李',
        age: 20
      }
    };
  },
  watch: {
    person: {
      handler(newValue, oldValue) {
        console.log(`从 ${oldValue} 变为 ${newValue}`);
      },
    },
  },
```

```
    methods: {
      handleChange() {
        this.person = {
          name: '小王',
          age: 26
        };
      },
    },
};
</script>

<template>
  <div>
    <p>姓名：{{ person.name }} -- 年龄：{{ person.age }}</p>
    <button @click="handleChange">更换人物</button>
  </div>
</template>
```

保存之后，单击“更换人物”按钮，person 对象会被替换为一个新的对象，即使没有使用 deep:true，这也会触发 watch 监听器的处理函数，并将新值和旧值作为参数输出到浏览器控制台。

3.7.4 监听对象中某个属性

【例 3-31】 监听对象中某个属性的变化，修改 App.vue，代码如下所示。

```
<script>
export default {
  data() {
    return {
      obj: {
        property: 'Hello'
      }
    };
  },
  watch: {
    'obj.property': function (newValue, oldValue) {
      console.log(`属性值从 ${oldValue}变为 ${newValue}`);
    }
  },
  methods: {
    updateProperty() {
      this.obj.property = 'World';
    }
  }
};
</script>

<template>
```

```
    <div>
      <p>对象属性值: {{ obj.property }}</p>
      <button @click="updateProperty">更新属性值</button>
    </div>
  </template>
```

在 watch 选项中,使用字符串形式的路径来指定要监听的对象属性,即'obj.property'。当 obj.property 的值发生变化时,监听器会触发并执行相关的处理函数。

当单击按钮更新属性值时,watch 监听器会捕获到 obj.property 的变化,并将新值和旧值输出到浏览器控制台,打印出属性值从 Hello 变为 World。

3.8 事件处理

事件处理是一种常见的交互方式,用于响应用户的操作或组件之间的通信。我们可以使用 v-on 指令(简写为@)监听 DOM 事件,并在事件触发时执行对应的函数,例如最常用的鼠标单击事件@click。

3.8.1 鼠标事件

1. click

click 事件是指当用户在一个元素上单击鼠标时触发的事件。具体而言,当用户按下鼠标左键并释放时,click 事件将被触发。

【例 3-32】 修改 App.vue,代码如下所示。

```
<script>
export default {
  methods: {
    handleClick() {
      console.log('Button clicked!');
    }
  }
}
</script>

<template>
  <div>
    <button @click="handleClick">Click me</button>
  </div>
</template>
```

在模板中使用 v-on:click(简写为@click)指令来绑定 handleClick 方法作为单击事件的处理函数。保存代码后,浏览器显示内容进行更新。当按钮被单击时,handleClick 方法会被调用,浏览器控制台会输出“Button clicked!”。

2. dblclick

dblclick 事件是指当鼠标双击一个元素时触发的事件。具体而言，当用户在一个元素上连续单击鼠标两次时，dblclick 事件将被触发。

【例 3-33】 修改 App.vue，代码如下所示。

```
<script>
export default {
  methods: {
    handleDoubleClick() {
      console.log("按钮被双击了!");
    },
  },
};
</script>

<template>
  <div>
    <button @dblclick = "handleDoubleClick">双击我</button>
  </div>
</template>
```

在模板中使用@dblclick 指令来绑定 handleDoubleClick 方法作为双击事件的处理函数。当按钮被双击时，handleDoubleClick 方法会被调用，控制台会输出“按钮被双击了!”。

3. mouseover

mouseover 事件是指当鼠标指针移动到一个元素上时触发的事件。具体而言，当鼠标指针从元素外部移动到元素内部时，mouseover 事件将被触发。

【例 3-34】 修改 App.vue，代码如下所示。

```
<script>
export default {
  methods: {
    handleMouseOver() {
      console.log('鼠标移入了按钮!');
    }
  }
}
</script>

<template>
  <div>
    <button @mouseover = "handleMouseOver">鼠标移入我</button>
  </div>
</template>
```

在模板中使用@mouseover 指令来绑定 handleMouseOver 方法作为鼠标移入事件的处理函数。当鼠标移入按钮时，handleMouseOver 方法会被调用，控制台会输出“鼠标移入了按钮!”。

4. mouseleave

mouseleave 事件是指当鼠标指针从元素上移出时触发的事件。具体而言，当鼠标指针移出一个元素或其子元素的范围时，mouseleave 事件将被触发。

【例 3-35】 修改 App.vue，代码如下所示。

```
<script>
export default {
  methods: {
    handleMouseLeave() {
      console.log('鼠标离开了按钮!');
    }
  }
}
</script>

<template>
  <div>
    <button @mouseleave="handleMouseLeave">鼠标离开我</button>
  </div>
</template>
```

在模板中使用@mouseleave 指令来绑定 handleMouseLeave 方法作为鼠标离开事件的处理函数。当鼠标离开按钮时，handleMouseLeave 方法会被调用，控制台会输出“鼠标离开了按钮!”。

5. 拖曳事件

拖曳事件包括 dragstart、dragenter、dragover、dragleave、dragend 和 drop。

(1) dragstart：当拖动操作开始时触发该事件。在该事件中，可以设置拖动数据和样式。

(2) dragenter：当被拖动的元素进入目标元素的范围内时触发该事件。可以在该事件中设置目标元素的样式，表示拖动元素进入了有效的放置区域。

(3) dragover：当被拖动的元素在目标元素上移动时持续触发该事件。在该事件中，可以阻止默认的拖动行为，并根据需要设置样式或执行其他操作。

(4) dragleave：当被拖动的元素离开目标元素的范围时触发该事件。可以在该事件中还原目标元素的样式，表示拖动元素已离开有效的放置区域。

(5) dragend：当拖动操作结束时触发该事件。在该事件中，可以清除拖动过程中设置的样式或其他操作。

(6) drop：当被拖动的元素在目标元素上释放时触发该事件。在该事件中，可以获取拖动数据并执行相应的操作，如将数据添加到目标元素中。

【例 3-36】 修改 App.vue，代码如下所示。

```
<script>
export default {
  methods: {
```

```
    handleDragStart(event) {
      event.dataTransfer.setData("text/plain", event.target.id);
    },
    handleDragEnter(event) {
      event.target.classList.add("drag-over");
    },
    handleDragOver(event) {
      event.preventDefault();
    },
    handleDragLeave(event) {
      event.target.classList.remove("drag-over");
    },
    handleDragEnd(event) {
      event.target.classList.remove("drag-over");
    },
    handleDrop(event) {
      event.preventDefault();
      const data = event.dataTransfer.getData("text/plain");
      console.log("拖曳的元素 ID:", data);
    },
  },
};
</script>

<template>
  <div>
    <div
      class="drag-box"
      draggable="true"
      id="drag-box"
      @dragstart="handleDragStart"
      @dragenter="handleDragEnter"
      @dragover="handleDragOver"
      @dragleave="handleDragLeave"
      @dragend="handleDragEnd"
    >
      拖曳我
    </div>
    <div
      class="drop-box"
      @dragover="handleDragOver"
      @dragenter="handleDragEnter"
      @dragleave="handleDragLeave"
      @drop="handleDrop"
    >
      放置区域
    </div>
  </div>
</template>
```

```
<style>
.drag-box {
  width: 100px;
  height: 100px;
  background-color: #f00;
  color: #fff;
  text-align: center;
  line-height: 100px;
  cursor: move;
}

.drop-box {
  width: 200px;
  height: 200px;
  background-color: #0f0;
  color: #fff;
  text-align: center;
  line-height: 200px;
}

.drag-over {
  border: 2px dashed #000;
}
</style>
```

这个示例展示了一个简单的拖曳功能。在模板中，有两个 元素，一个用作拖曳源(drag-box)，另一个用作放置区域(drop-box)。通过设置 draggable="true"，将拖曳源设置为可拖曳的。

在拖曳源的元素上绑定了 dragstart、dragenter、dragover、dragleave 和 dragend 事件，并在放置区域上绑定了 dragover、dragenter、dragleave 和 drop 事件。

handleDragStart 方法在拖曳开始时被调用，将拖曳元素的 ID 设置为传输的数据。handleDragEnter、handleDragOver 和 handleDragLeave 方法分别在拖曳元素进入、在其上方移动和离开时被调用，通过添加或移除 drag-over 类来修改样式。handleDragEnd 方法在拖曳结束时被调用，移除 drag-over 类。handleDrop 方法在放置区域内放置拖曳元素时被调用，阻止默认的放置行为，并从事件中获取传输的数据。

在样式部分，定义了拖曳源和放置区域的样式，以及当拖曳元素在放置区域上方时显示虚线边框的样式。

3.8.2 键盘事件

1. keydown

keydown 是指键盘上的按键被按下的事件。当用户按下一个键时，keydown 事件会被触发。

【例 3-37】 修改 App.vue，代码如下所示。

```
<script>
export default {
  data() {
    return {
      text: "",
      keyPressed: ""
    };
  },
  methods: {
    handleKeyDown(event) {
      this.keyPressed = event.key;
    }
  }
};
</script>

<template>
  <div>
    <input type="text" v-model="text" @keydown="handleKeyDown" />
    <p>按下的键：{{ keyPressed }}</p>
  </div>
</template>
```

在这个示例中，有一个文本输入框，当用户在文本输入框中按下键盘上的任意键时，触发 keydown 事件调用 handleKeyDown 方法，将按下的键的值赋给 keyPressed 变量。

2. keyup

keyup 是指键盘上的按键被释放的事件。当用户按下一个键并释放时，keyup 事件会被触发。

【例 3-38】 修改 App.vue，代码如下所示。

```
<script>
export default {
  data() {
    return {
      text: "",
      keyPressed: "",
    };
  },
  methods: {
    handleKeyUp(event) {
      this.keyPressed = event.key;
    },
  },
};
</script>

<template>
```

```
    <div>
      <input
        type = "text"
        v-model = "text"
        @keyup = "handleKeyUp"
        placeholder = "Type something..."
      />
      <p>Pressed Key: {{ keyPressed }}</p>
    </div>
  </template>
```

在这个示例中，有一个文本输入框，当用户释放键盘上的任意键时，keyup 事件会触发 handleKeyUp 方法。该方法将被调用，并将释放的键的值赋给 keyPressed 变量。

3. keypress

keypress 是一个键盘事件，在用户按下键盘上的键时触发。它主要用于处理按下字符键（包括字母、数字和符号）的情况，但不包括功能键、控制键或特殊键。

【例 3-39】 修改 App.vue，代码如下所示。

```
<script>
export default {
  methods: {
    handleKeyPress(event) {
      console.log("Key pressed:", event.key);
    },
  },
};
</script>

<template>
  <input type = "text" @keypress = "handleKeyPress" />
</template>
```

在这个示例中，有一个文本输入框，通过使用@keypress 指令绑定 handleKeyPress 方法到 keypress 事件上。当用户在文本输入框中按下键盘上的键时，handleKeyPress 方法会被调用。在该方法中，通过 event 参数获取按下的键的信息，并使用 console.log 打印出被按下的键的值。

3.8.3　焦点事件

1. focus

focus 事件用于监听元素获得焦点的情况。当元素被单击或通过程序获得焦点时，focus 事件将被触发。

【例 3-40】 修改 App.vue，代码如下所示。

```
<script>
export default {
```

```
    data() {
      return {
        isFocused: false,
      };
    },
    methods: {
      handleFocus() {
        this.isFocused = true;
      },
    },
  };
  </script>

  <template>
    <div>
      <input type="text" @focus="handleFocus" />
      <p>{{ isFocused ? "Input is focused" : "Input is not focused" }}</p>
    </div>
  </template>
```

在上面的示例中，在<input>元素上使用@focus 指令将 handleFocus 方法绑定到 focus 事件上。当输入框获得焦点时，handleFocus 方法将被调用，将 isFocused 属性设置为 true。

在模板中，根据 isFocused 属性的值来显示不同的文本。如果输入框获得焦点，显示 Input is focused，否则显示 Input is not focused。

通过这个示例，可以动态地跟踪输入框是否获得了焦点，并在界面上进行相应的展示或处理。

2. blur

blur 事件在元素失去焦点时触发，它是与 focus 事件相对应的事件。当用户将焦点从一个元素移开时，会触发该元素的 blur 事件。

【例 3-41】 修改 App.vue，代码如下所示。

```
  <script>
  export default {
    data() {
      return {
        isFocused: false,
      };
    },
    methods: {
      handleFocus() {
        this.isFocused = true;
      },
      handleBlur() {
        this.isFocused = false;
      },
    },
  };
```

```
</script>

<template>
  <div>
    <input type="text" @focus="handleFocus" @blur="handleBlur" />
    <p>{{ isFocused ? "Input is focused" : "Input is not focused" }}</p>
  </div>
</template>
```

在上面的示例中，使用 data 选项定义了一个名为 isFocused 的响应式数据，默认值为 false。还定义了两个方法 handleFocus 和 handleBlur，分别用于处理输入框获得焦点和失去焦点的事件。

在模板中，使用@focus 和@blur 指令将这两个方法绑定到输入框的 focus 和 blur 事件上。当输入框获得焦点时，handleFocus 方法将被调用，将 isFocused 的值设置为 true；当输入框失去焦点时，handleBlur 方法将被调用，将 isFocused 的值设置为 false。

通过{{ isFocused ? "Input is focused" : "Input is not focused" }}，我们可以根据 isFocused 的值来动态显示不同的文本。如果输入框获得焦点，则显示 Input is focused；否则显示 Input is not focused。

在示例中，我们可以根据输入框的焦点状态来改变界面的展示内容，实现与焦点相关的交互效果。

3.8.4 表单事件

1. change

change 事件用于处理表单元素的值发生改变时触发的事件。它适用于文本输入框、复选框、单选框和下拉列表等表单元素。

当用户对表单元素进行修改并且失去焦点时，会触发 change 事件。例如，当用户在文本输入框中输入文本并按下回车键或者单击其他地方时，change 事件会被触发。

【例 3-42】 修改 App.vue，代码如下所示。

```
<script>
export default {
  data() {
    return {
      message: ""
    };
  },
  methods: {
    handleChange(event) {
      console.log("新的值: ", event.target.value);
    }
  }
};
</script>
```

```
<template>
  <div>
    <input type="text" v-model="message" @change="handleChange" />
    <p>当前的值：{{ message }}</p>
  </div>
</template>
```

在上面的示例中，绑定了一个文本输入框的 change 事件，并将输入框的值绑定到 message 属性上。当用户修改输入框的值并且失去焦点时，handleChange 方法会被调用，并在控制台中打印出新的值。

通过使用 change 事件，我们可以监听表单元素值的变化，并在变化时执行相应的逻辑。这对于实时响应用户的输入或进行表单验证非常有用。

2. input

input 事件用于处理实时监测输入框的值变化的事件。与 change 事件不同，input 事件实时地反映输入框值的变化，而不是在失去焦点时才触发。这实现了实时响应用户的输入并进行相应的处理，如实时搜索、动态计算等。

【例 3-43】 修改 App.vue，代码如下所示。

```
<script>
export default {
  data() {
    return {
      message: ""
    };
  },
  methods: {
    handleInput(event) {
      console.log("当前输入的值：", event.target.value);
    }
  }
};
</script>
<template>
  <div>
    <input type="text" v-model="message" @input="handleInput" />
    <p>当前的值：{{ message }}</p>
  </div>
</template>
```

在上面的示例中，绑定了一个文本输入框的 input 事件，并将输入框的值绑定到 message 属性上。每当用户输入或修改输入框的内容时，handleInput 方法会被调用，并在控制台中打印出当前输入的值。

3. submit

submit 事件用于处理表单提交的事件。当用户在表单中单击提交按钮或按下回车键时，submit 事件会触发。

【例 3-44】 修改 App.vue，代码如下所示。

```
<script>
export default {
  data() {
    return {
      name: "",
      submitted: false,
    };
  },
  methods: {
    handleSubmit(event) {
      event.preventDefault(); //阻止表单的默认提交行为

      //执行提交逻辑
      this.submitted = true;
    },
  },
};
</script>

<template>
  <form @submit="handleSubmit">
    <label for="name">姓名：</label>
    <input type="text" id="name" v-model="name" />
    <button type="submit">提交</button>
  </form>
  <p v-if="submitted">已提交：{{ name }}</p>
</template>
```

上述示例中有一个简单的表单，包含一个输入框和一个提交按钮。当用户在输入框中输入姓名并单击提交按钮或按下回车键时，submit 事件会被触发，并调用 handleSubmit 方法处理提交逻辑。

在 handleSubmit 方法中，首先调用 event.preventDefault()阻止表单的默认提交行为，然后将 submitted 属性设置为 true，表示表单已提交。在模板中，使用 v-if 指令根据 submitted 属性的值显示提交成功的消息。

通过该示例，可以学到如何使用 submit 事件来处理表单的提交，并在提交时执行自定义的逻辑。

3.8.5　滚动事件

滚动事件是指在网页或元素内容区域滚动时触发的事件。在前端开发中，滚动事件非常常见，常用于监听用户滚动页面或滚动某个元素的操作。

【例 3-45】 修改 App.vue，代码如下所示。

```
<script>
export default {
  methods: {
    handleScroll(event) {
      const container = event.target;
      const scrollHeight = container.scrollHeight;
      const scrollTop = container.scrollTop;
      const clientHeight = container.clientHeight;

      if (scrollHeight - scrollTop === clientHeight) {
        //当滚动到底部时执行相应逻辑
        console.log("已滚动到底部");
      }
    },
  },
};
</script>

<template>
  <div class="scroll-container" @scroll="handleScroll">
    <div class="scroll-content">
      <!-- 这里是滚动内容 -->
    </div>
  </div>
</template>

<style>
.scroll-container {
  width: 400px;
  height: 300px;
  overflow: auto;
}

.scroll-content {
  height: 800px;
  background-color: #f0f0f0;
}
</style>
```

上述示例创建了一个滚动容器 scroll-container，并给容器绑定了滚动事件@scroll。当滚动容器的内容发生滚动时，handleScroll 方法将被调用。

在 handleScroll 方法中，通过事件对象 event 获取了滚动容器的滚动高度（scrollHeight）、滚动的位置（scrollTop）以及可见区域的高度（clientHeight）。通过这些信息，可以判断滚动是否到达了底部（scrollHeight－scrollTop ＝＝＝ clientHeight），并执行相应的逻辑。在示例中，当滚动到底部时，控制台会打印出“已滚动到底部”的消息。

3.8.6　文本相关事件

copy、cut 和 paste 是常见的与文本内容相关的操作。

(1) copy：复制操作可以通过监听 copy 事件实现。当用户进行复制操作时，copy 事件会被触发。

(2) cut：剪切操作可以通过监听 cut 事件实现。当用户进行剪切操作时，cut 事件会被触发。

(3) paste：粘贴操作可以通过监听 paste 事件实现。当用户进行粘贴操作时，paste 事件会被触发。

【例 3-46】 修改 App.vue，代码如下所示。

```
<script>
export default {
  data() {
    return {
      text: "",
    };
  },
  methods: {
    handleCopy(event) {
      console.log("复制操作");
    },
    handleCut(event) {
      console.log("剪切操作");
    },
    handlePaste(event) {
      console.log("粘贴操作");
    },
    copyText() {
      navigator.clipboard
        .writeText(this.text)
        .then(() => {
          console.log("文本已复制到剪贴板");
        })
        .catch((error) => {
          console.error("复制文本失败:", error);
        });
    },
    cutText() {
      navigator.clipboard
        .writeText(this.text)
        .then(() => {
          console.log("文本已剪切到剪贴板");
          this.text = "";
        })
        .catch((error) => {
```

```
            console.error("剪切文本失败:", error);
          });
      },
      pasteText() {
        navigator.clipboard
          .readText()
          .then((text) => {
            console.log("从剪贴板粘贴的文本:", text);
            this.text = text;
          })
          .catch((error) => {
            console.error("粘贴文本失败:", error);
          });
      },
    },
  };
  </script>

  <template>
    <div>
      <input
        type="text"
        v-model="text"
        @copy="handleCopy"
        @cut="handleCut"
        @paste="handlePaste"
      />
      <button @click="copyText">复制</button>
      <button @click="cutText">剪切</button>
      <button @click="pasteText">粘贴</button>
    </div>
  </template>
```

上述示例使用了 navigator.clipboard 对象访问剪贴板。navigator.clipboard.writeText 方法用于将文本写入剪贴板，而 navigator.clipboard.readText 方法用于从剪贴板中读取文本。

注意：使用 navigator.clipboard API 需要在安全的上下文环境中，如在 HTTPS 网站或本地开发环境中。另外，不同浏览器对 navigator.clipboard 的支持程度可能会有所不同，如 Chrome 浏览器在使用粘贴功能时，会弹出提示框询问是否允许执行该操作。

3.8.7 事件传参

【例 3-47】 给方法传参是个常见的业务场景。修改 App.vue，代码如下所示。

```
  <script>
  export default {
    methods: {
      handleClick(message) {
```

```
        console.log(message); //输出 'hello'
      },
    },
  };
  </script>

  <template>
    <div>
      <button @click="handleClick('hello')">按钮</button>
    </div>
  </template>
```

上述示例定义了一个名为 handleClick 的事件处理方法，它接收一个参数 message。当按钮被单击时，通过内联方法@click 将参数“hello”传递给 handleClick 方法。handleClick 方法会接收该参数，并在浏览器控制台中将值打印出来。

3.8.8　事件修饰符

在处理事件时调用 event.preventDefault()或 event.stopPropagation()是很常见的。在介绍拖曳事件时便用到了 event.preventDefault()。

event.preventDefault()是一个用于阻止事件的默认行为的方法。当事件发生时，浏览器会执行一些默认的操作，如单击链接会跳转到链接的地址，提交表单会刷新页面等。通过调用 event.preventDefault()可以阻止这些默认行为的发生。

event.stopPropagation()是一个用于停止事件传播的方法。当事件被触发时，它会沿着 DOM 树从目标元素向上冒泡或从根元素向下捕获。这种传播方式允许事件在 DOM 树中的多个元素之间进行交互和响应。通过调用 event.stopPropagation()可以阻止事件的进一步传播，从而停止事件从目标元素向上冒泡或从根元素向下捕获。

虽然可以直接在方法内调用 event.preventDefault()或 event.stopPropagation()，但 Vue.js 提供了事件修饰符（修饰符是用“.”表示的指令后缀，如.stop、.prevent、.self、.capture、.once 和.passive 等），可以让我们更专注于数据逻辑而不用去处理 DOM 事件的细节。

1. .prevent

.prevent 修饰符实现了 event.preventDefault()的功能。下面先展示 event.preventDefault()阻止事件的默认行为示例。

【例 3-48】 修改 App.vue，代码如下所示。

```
<script>
export default {
  methods: {
    handleClick(event) {
      event.preventDefault(); //阻止链接的默认行为
      console.log("链接被单击了");
    },
```

```
  },
};
</script>

<template>
  <a href="https://www.example.com" @click="handleClick">单击跳转</a>
</template>
```

在上述示例中，当链接被单击时，通过调用 handleClick 方法中的 event.preventDefault()，阻止了链接的默认行为，即不会跳转到指定的 URL。而在控制台中会打印出“链接被单击了”的消息。通过使用 event.preventDefault()，我们可以自定义处理链接的单击行为，而不会触发默认的页面跳转。

【例 3-49】 使用修饰符达到一样的效果，修改 App.vue，代码如下所示。

```
<script>
export default {
  methods: {
    handleClick(event) {
      console.log("链接被单击了");
    },
  },
};
</script>

<template>
  <a href="https://www.example.com" @click.prevent="handleClick">单击跳转</a>
</template>
```

2. .stop

在 DOM 中，事件传播过程涉及 3 个阶段，即捕获阶段（capturing phase）、目标阶段（target phase）和冒泡阶段（bubbling phase）。

（1）捕获阶段：事件从根节点（即 window 对象）向下传播，直到达到触发事件的目标元素的父级元素。

（2）目标阶段：当事件到达目标元素时，它会在目标元素上触发相应的事件处理函数。这个阶段只包含目标元素自身的事件处理。

（3）冒泡阶段：事件从目标元素开始向上传播，直到到达根节点。这个阶段像水里的气泡一样从下向上传播。

【例 3-50】 先展示事件冒泡行为。修改 App.vue，代码如下所示。

```
<script>
export default {
  methods: {
    handleOuterClick(event) {
      console.log("Outer div clicked");
    },
    handleInnerClick(event) {
```

```
        console.log("Inner div clicked");
      },
      handleButtonClick(event) {
        console.log("Button clicked");
      },
    },
};
</script>

<template>
  <div @click="handleOuterClick">
    <div @click="handleInnerClick">
      <button @click="handleButtonClick">按钮</button>
    </div>
  </div>
</template>
```

单击按钮，浏览器控制台打印“Button clicked”“Inner div clicked”“Outer div clicked”，说明单击按钮时，在捕获阶段执行 handleButtonClick 函数，然后在冒泡阶段依次执行 handleInnerClick 和 handleOuterClick 函数。

修饰符. stop 可以阻止事件冒泡。修改 App. vue 的模板区域，给按钮的单击事件 handleButtonClick 添加. stop 修饰符，代码如下所示。

```
<template>
  <div @click="handleOuterClick">
    <div @click="handleInnerClick">
      <button @click.stop="handleButtonClick">按钮</button>
    </div>
  </div>
</template>
```

保存代码后，单击浏览器页面的按钮，浏览器控制台只打印“Button clicked”。handleInnerClick 和 handleOuterClick 两个冒泡事件未被执行。

3. .capture

.capture 修饰符用于指定事件监听函数在事件捕获阶段进行处理，而不是默认的事件冒泡阶段。

【例 3-51】 修改 App. vue，代码如下所示。

```
<script>
export default {
  methods: {
    handleDivClick(event) {
      console.log("Div clicked");
    },
    handleButtonClick(event) {
      console.log("Button clicked");
    },
```

```
  },
};
</script>

<template>
  <div @click.capture="handleDivClick">
    <button @click.capture="handleButtonClick">按钮</button>
  </div>
</template>
```

在此示例中，我们在 元素和<button>元素上的单击事件绑定中使用.capture修饰符。当单击按钮时，相应的单击事件处理函数handleDivClick和handleButtonClick会依次被调用。可以看到浏览器控制台打印出“Div clicked”和“Button clicked”。

通过使用.capture修饰符，事件会在捕获阶段触发，而不是默认的冒泡阶段。

4. .self

.self修饰符用于限制事件只在触发事件的元素自身上触发，而不会在其他元素上触发。

【例3-52】 修改App.vue，代码如下所示。

```
<script>
export default {
  methods: {
    handleOuterClick(event) {
      console.log("Outer div clicked");
    },
    handleInnerClick(event) {
      console.log("Inner div clicked");
    },
    handleButtonClick(event) {
      console.log("Button clicked");
    },
  },
};
</script>

<template>
  <div @click="handleOuterClick">
    <div @click.self="handleInnerClick" class="box">
      <button @click="handleButtonClick">按钮</button>
    </div>
  </div>
</template>

<style>
.box{
  width: 200px;
  height: 200px;
```

```
    background-color: bisque;
  }
</style>
```

handleInnerClick 是绑定在内部 元素上的单击事件处理函数，并使用.self 修饰符。这意味着只有当单击内部 元素本身时，才会触发该事件处理函数。

单击按钮，浏览器控制台打印“Button clicked”“Outer div clicked”，说明在冒泡阶段只执行了 handleOuterClick 函数，而使用.self 修饰符的 handleInnerClick 函数没有被触发。

5. .once

.once 修饰符用于指定事件只能被触发一次。当使用.once 修饰符绑定事件处理函数时，该事件处理函数只会在第一次触发事件时执行，随后的同类型事件将不再触发该处理函数。

【例 3-53】 修改 App.vue，代码如下所示。

```
<script>
export default {
  methods: {
    handleClick() {
      console.log("按钮被单击了!");
    },
  },
};
</script>

<template>
  <div>
    <button @click.once="handleClick">按钮</button>
  </div>
</template>
```

在上面的示例中，我们在按钮上使用.once 修饰符来绑定 handleClick 方法作为单击事件的处理函数。当按钮被单击时，handleClick 方法只会被执行一次，即只在首次单击时输出“按钮被单击了!”。随后的单击事件将不再触发该处理函数，即浏览器控制台只会打印一次“按钮被单击了!”。

6. .passive

.passive 修饰符是用于优化滚动事件的性能的一种技巧。告诉浏览器该事件处理函数不会阻止默认行为，浏览器可以在滚动事件被触发时立即进行滚动处理，提高滚动的响应性能。

修饰符可以链式书写，例如<a @click.stop.prevent="doThat">表示阻止事件冒泡并且阻止浏览器默认行为。

3.8.9 按键修饰符

按键修饰符是在 Vue.js 中用于处理键盘按键事件的一种方式。它们允许你指定只有

在特定按键被按下时才触发事件处理函数。

1. 按键别名

Vue.js 为一些常用的按键提供了别名。

(1).enter：当回车键被按下时触发事件。

(2).tab：当 Tab 键被按下时触发事件。

(3).delete：捕获 Delete 和 Backspace 两个按键。

(4).esc：当 Esc 键被按下时触发事件。

(5).space：当空格键被按下时触发事件。

(6).up：当上箭头键被按下时触发事件。

(7).down：当下箭头键被按下时触发事件。

(8).left：当左箭头键被按下时触发事件。

(9).right：当右箭头键被按下时触发事件。

【例 3-54】 以.enter 为例，修改 App.vue，代码如下所示。

```
<script>
export default {
  methods: {
    handleEnterKey(event) {
      console.log("Enter key pressed");
      //执行其他逻辑
    },
  },
};
</script>

<template>
  <input type="text" @keydown.enter="handleEnterKey" />
</template>
```

在这个示例中，当用户在<input>输入框中按下回车键时，handleEnterKey 函数会被调用，并在控制台输出“Enter key pressed”。

【例 3-55】 以.up 为例，修改 App.vue，代码如下所示。

```
<script>
export default {
  methods: {
    handleUpKey(event) {
      console.log("up");
      //执行其他逻辑
    },
  },
};
</script>

<template>
```

```
    <input type="text" @keydown.up="handleUpKey" />
  </template>
```

在这个示例中，当用户按下键盘上的向上箭头键时，handleUpKey 函数会被调用，并在控制台输出“up”。

2. 系统按键修饰符

你可以使用以下系统按键修饰符来触发鼠标或键盘事件监听器：.ctrl、.alt、.shift、.meta。在 Mac 键盘上，meta 是 Command 键(⌘)。在 Windows 键盘上，meta 键是 Windows 键(⊞)。

【例 3-56】 以.ctrl 为例，用户按下 Ctrl＋A 键，事件会被触发。修改 App.vue，代码如下所示。

```
<script>
export default {
  methods: {
    handleCtrlKeyDown(event) {
      if (event.key === "a") {
        console.log("Ctrl + a pressed");
        //执行其他逻辑
      }
    },
  },
};
</script>

<template>
  <div>
    <input type="text" @keydown.ctrl="handleCtrlKeyDown" />
  </div>
</template>
```

3.9　类与样式绑定

在处理元素的 CSS class 列表和内联样式时，数据绑定是一个常见的需求场景。在 Vue.js 中，我们可以使用 v-bind 指令将 class 和 style 属性与动态字符串进行绑定，就像处理其他属性一样。

3.9.1　类绑定

1. 绑定对象

【例 3-57】 修改 App.vue，代码如下所示。

```
<script>
export default {
  data() {
    return {
```

```
        isRed: true,
      };
    },
  };
  </script>

  <template>
    <div :class="{ red: isRed }">CSS class 绑定示例</div>
  </template>

  <style scoped>
  .red {
    background-color: red;
  }
  </style>
```

在模板中，使用:class(v-bind: class 的简写)指令将 isRed 属性与 CSS class 绑定。对象语法{ red:isRed }表示当 isRed 为 true 时，red 类将被应用到< div >元素上。这样，当 isRed 为 true 时，有

```
<div :class="{ red: isRed }">CSS class 绑定示例</div>
```

可以看成

```
<div class="red">CSS class 绑定示例</div>
```

【例 3-58】 :class 指令也可以和一般的 class 类名共存，修改 App.vue，代码如下所示。

```
<script>
export default {
  data() {
    return {
      isRed: true,
    };
  },
};
</script>

<template>
  <div class="white" :class="{ red: isRed }">CSS class 绑定示例</div>
</template>

<style scoped>
.red {
  background-color: red;
}
.white{
  color: white;
}
</style>
```

当 isRed 为 true 时，有

```
<div class="white" :class="{ red: isRed }">CSS class 绑定示例</div>
```

渲染的结果会是

```
<div class="white red">CSS class 绑定示例</div>
```

2. 绑定数组

如果想动态绑定一个包含多个 CSS class 的数组，可以使用数组语法进行 CSS class 绑定。

【例 3-59】 修改 App.vue，代码如下所示。

```
<script>
export default {
  data() {
    return {
      classArray: ['white', 'red']
    };
  }
};
</script>

<template>
  <div :class="classArray">CSS class 绑定示例</div>
</template>

<style scoped>
.white {
  color: white;
}

.red {
  background-color: red;
}
</style>
```

在模板中，我们使用：class 指令将 classArray 数组作为绑定值。这样， 元素会动态地应用 classArray 数组中的所有 CSS class。

```
<div :class="classArray">CSS class 绑定示例</div>
```

渲染结果会是

```
<div class="'white red">CSS class 绑定示例</div>
```

如果想在数组中有条件地渲染某个 class，可以使用三元表达式。

【例 3-60】 修改 App.vue，代码如下所示。提示：数组外用双引号包裹，white、red 等字符串用单引号包裹。

```
<script>
export default {
```

```
  data() {
    return {
      isActive: true,
    };
  },
};
</script>

<template>
  <div :class="['white', isActive ? 'red' : 'blue']">CSS class 绑定示例</div>
</template>

<style scoped>
.white {
  color: white;
}

.red {
  background-color: red;
}

.blue {
  background-color: blue;
}
</style>
```

在模板中，使用：class 指令将包含 white 和动态选择的 red 或 blue 的 CSS class 数组进行绑定。根据 isActive 的值，如果为 true，则选择 red 类，否则选择 blue 类。

通过这种方式，元素的 CSS class 会根据 isActive 属性的值动态切换，实现了根据条件选择不同的样式。

如果只是想根据 isActive 的值来判断是否为 red 时，也可以使用数组中嵌套对象来简化。

【例 3-61】 修改 App.vue，代码如下所示。

```
<script>
export default {
  data() {
    return {
      isActive: true,
    };
  },
};
</script>

<template>
  <div :class="['white', { 'red': isActive }]">CSS class 绑定示例</div>
</template>
```

```
<style scoped>
.white {
  color: white;
}

.red {
  background-color: red;
}

.blue {
  background-color: blue;
}
</style>
```

在模板中，使用:class 指令将包含 white 和一个对象的 CSS class 数组进行绑定。对象中的键是 CSS class 的名称，而值是一个布尔表达式，表示在对应的条件下是否应用该 CSS class。在这里，我们根据 isActive 的值决定是否应用 red 类。

3.9.2　绑定内联样式

在上述例子中，使用属性动态绑定类名，也可以使用属性动态绑定内联样式。

1. 绑定对象

如果想动态地绑定一个包含多个 CSS 样式属性和值的对象到元素的内联样式，可以使用对象语法进行内联样式的绑定。

【例 3-62】　修改 App.vue，代码如下所示。

```
<script>
export default {
  data() {
    return {
      styleObject: {
        color: 'red',
        fontSize: '20px',
        backgroundColor: 'blue'
      }
    };
  }
};
</script>

<template>
  <div :style="styleObject">内联样式绑定示例</div>
</template>
```

在模板中，使用:style(v-bind :style 的简写)指令将 styleObject 对象作为绑定值。这样，元素的内联样式将根据 styleObject 对象中定义的属性和值进行渲染。在这个例子中，文字颜色将被设置为红色，字体大小为 20 像素，背景颜色为蓝色。

通过绑定一个包含多个 CSS 样式属性和值的对象，可以根据需要动态修改元素的内联样式，实现更灵活的样式控制。在浏览器控制台的 Elements 选项中可查看效果，如图 3-8 所示。

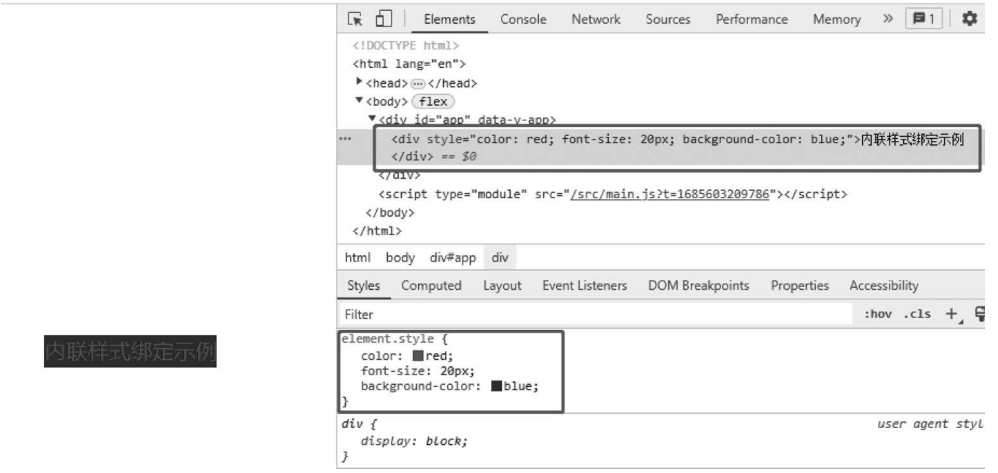

图 3-8　绑定内联样式

2. 绑定数组

还可以给:style 绑定一个包含多个样式对象的数组。这些对象会被合并后渲染到同一元素上。

【例 3-63】 修改 App.vue，代码如下所示。

```
<script>
export default {
  data() {
    return {
      styleObject1: {
        color: 'red',
        fontSize: '20px'
      },
      styleObject2: {
        backgroundColor: 'blue'
      }
    };
  }
};
</script>

<template>
  <div :style="[styleObject1, styleObject2]">内联样式绑定示例</div>
</template>
```

在脚本中，定义了两个对象，即 styleObject1 和 styleObject2，它们分别包含了一组 CSS 样式属性和值。

在模板中，使用数组[styleObject1，styleObject2]作为:style 指令的绑定值。这样，元素的内联样式将根据数组中的所有对象进行渲染。

在这个例子中，文字颜色将被设置为红色，字体大小为 20 像素，背景颜色将被设置为蓝色。通过绑定一个包含多个 CSS 样式的数组，我们可以同时应用多个样式对象，实现更复杂的内联样式控制。

3.10 模板引用

虽然 Vue.js 的声明性渲染模型为你抽象了大部分对 DOM 的直接操作，但在某些情况下，我们仍然需要直接访问底层 DOM 元素。要实现这一点，可以使用特殊的 ref 属性。通过在元素上添加 ref 属性并为其赋予一个名称，可以在组件中访问到这个 DOM 元素的引用。

【例 3-64】 使用 ref 属性来获取对输入框元素的直接引用。修改 App.vue，代码如下所示。

```
<script>
export default {
  created() {
    console.log("created", this.$refs.myInput); //created undefined
  },
  mounted() {
    console.log("mounted", this.$refs.myInput); //mounted <input type="text">
  },
  methods: {
    handleClick() {
      const inputValue = this.$refs.myInput.value;
      console.log("输入框的值: ", inputValue);
    },
  },
};
</script>

<template>
  <div>
    <input ref="myInput" type="text" />
    <button @click="handleClick">获取输入框值</button>
  </div>
</template>
```

在<script>部分，使用 methods 定义了一个方法 handleClick()。该方法通过 this.$refs.myInput 获取到输入框元素的引用，然后获取输入框的值并输出到控制台。

在<template>部分，使用<input>元素定义了一个输入框，并通过 ref 属性给它指定了一个名称为 myInput 的引用。使用<button>元素定义了一个按钮，通过@click 监听按钮的单击事件，并绑定到 handleClick 方法上。当按钮被单击时，调用 handleClick 方法，获取

输入框的值并输出到控制台。

需要注意的是，ref 的引用只在组件实例被挂载后才可用。可以看到，在 created 生命周期钩子函数中，打印 this.$refs.myInput 的值为 undefined；在 mounted 生命周期钩子函数中，打印 this.$refs.myInput 的值为 input 元素，如图 3-9 所示。

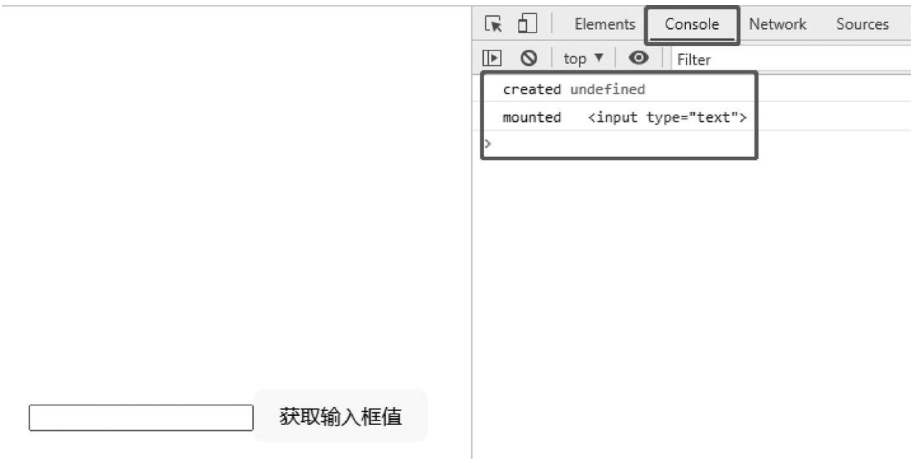

图 3-9 created 与 mounted 生命周期中 $refs 打印值

因此，我们常在 created 生命周期钩子函数中获取接口数据，在 mounted 生命周期钩子函数中获取 DOM 的值。如果想在 created 生命周期钩子函数中获取到 DOM 的值，则可以借助 this.$nextTick。$nextTick 的作用是在下次 DOM 更新循环结束之后，执行指定的回调函数。

将上述示例 created 中的代码修改为

```
created() {
  this.$nextTick(() => {
    console.log("created", this.$refs.myInput); //created <input type="text">
  });
},
```

保存之后，查看浏览器控制面板，如图 3-10 所示。

可以看到 created 生命周期在 this.$nextTick 回调函数中能打印出 DOM 的值，但执行时机在 mounted 的后面。ref 除了获取 DOM 元素，也可以获取组件的实例，用来调用子组件的方法，这会在 4.1 节组件通信中详细介绍。

拓展：this.$nextTick 中的回调函数是一个箭头函数()=>{}，箭头函数的特点如下。

(1) 简洁的语法：箭头函数可以使用更简洁的语法来定义函数，省略了 function 关键字。

(2) 隐式返回：当箭头函数的函数体只有一行时，大括号{}可以省略它会隐式返回这一行的结果，不需要使用 return 关键字。

(3) 没有自己的 this 值：箭头函数没有自己的 this 值，它继承了外层作用域的 this 值。

图 3-10 nextTick 的使用

(4) 没有 arguments 对象：箭头函数没有自己的 arguments 对象，但可以使用剩余参数获取函数的参数列表。

3.11 组件基础

组件允许我们将 UI 划分为独立的、可重用的部分，并且可以对每个部分进行单独的业务编写。之前我们已经接触过组件，src/components/HelloWorld.vue 文件就是一个单文件组件。单文件组件使用.vue 作为文件扩展名，包含三个主要部分：模板(template)、脚本(script)和样式(style)。每个部分都有自己的作用，可以独立地编写和修改。

3.11.1 定义与使用一个组件

【例 3-65】 修改 App.vue，代码如下所示。

```
<script>
import HelloWorld from "./components/HelloWorld.vue";
export default {
  components: {
    HelloWorld,
  },
};
</script>

<template>
  <HelloWorld />
</template>
```

在脚本部分，通过 import 语句导入了名为 HelloWorld 的组件，并在 components 选项

中注册了它。在模板部分，使用了< HelloWorld />渲染 HelloWorld 组件。

提示：确保你的文件路径和组件名称与实际的情况相匹配，并确保已经正确导入和注册了组件。

修改 HelloWorld.vue 文件，代码如下所示。

```
<script>
export default {
  data() {
    return {
      message: "Hello, Vue!",
    };
  },
  methods: {
    increment() {
      this.message += "!";
    },
  },
};
</script>

<template>
  <div>
    <h1>{{ message }}</h1>
    <button @click="increment">增加</button>
  </div>
</template>

<style scoped>
h1 {
  color: red;
}
button {
  background-color: blue;
  color: white;
}
</style>
```

在脚本部分，使用 data 方法定义了一个 message 数据属性，并初始化为"Hello, Vue!"。在 methods 对象中定义了一个 increment 方法，当按钮被单击时，会将 message 的值追加一个感叹号。

在模板部分，使用了插值语法{{ message }}将 message 的值显示在< h1 >标签中，并将 increment 方法绑定到按钮的单击事件。

在样式部分，使用 scoped 属性确保样式只适用于当前组件。h1 标签的颜色被设置为红色，按钮的背景色为蓝色，文字颜色为白色。

这个组件的作用是在页面上显示一个标题和一个按钮，并且每次单击按钮时，在标题的末尾添加一个感叹号。保存 HelloWorld.vue 文件，可在页面中查看效果。

3.11.2　动态组件

动态组件是 Vue.js 中一种用于动态切换和渲染组件的机制。它允许根据应用程序的状态或条件，动态地选择要渲染的组件，并在需要时进行切换。

【例 3-66】 通过设置< component >组件的 v-bind：is 属性，动态地指定要渲染的组件。在 components 文件夹下新建 ComponentA.vue，代码如下所示。

```
<template>
    组件 A
</template>
```

保存完 ComponentA.vue 之后，在 components 文件夹下新建 ComponentB.vue，代码如下所示。

```
<template>
    组件 B
</template>
```

保存完 ComponentB.vue 之后，修改 App.vue，代码如下所示。

```
<script>
import ComponentA from "./components/ComponentA.vue";
import ComponentB from "./components/ComponentB.vue";

export default {
  data() {
    return {
      currentComponent: "ComponentA",
    };
  },
  methods: {
    toggleComponent() {
      this.currentComponent =
        this.currentComponent === "ComponentA" ? "ComponentB" : "ComponentA";
    },
  },
  components: {
    ComponentA,
    ComponentB,
  },
};
</script>

<template>
  <div>
    <button @click="toggleComponent">切换组件</button>
    <component :is="currentComponent"></component>
  </div>
</template>
```

在模板中使用< component >组件，并将：is 绑定到 currentComponent 数据属性，可以根据数据的变化动态地渲染不同的组件。单击“切换组件”按钮，使用 toggleComponent 方法修改 currentComponent 的值，从而在 ComponentA 和 ComponentB 之间进行切换。

本章小结

本章探讨了以下主题：

（1）MVVM 模式的概述与优点：了解了 MVVM 模式的基本概念。

（2）数据选项 data 与插值：学习了如何在 Vue.js 中使用 data 选项来定义组件的数据，并将数据通过插值绑定到模板中进行渲染。

（3）方法选项 methods：介绍了在 Vue.js 组件中定义方法的方法，并在模板中调用这些方法来触发相应的逻辑操作。

（4）选项式 API 生命周期：了解了 Vue.js 组件的生命周期钩子函数，它们提供了在组件不同阶段执行特定操作的机会。

（5）基本指令：学习了 Vue.js 中一些常用的指令，例如 v-if、v-for、v-bind 和 v-on，它们分别用于控制元素的显示与隐藏、循环渲染、属性绑定和事件处理。

（6）计算属性选项 computed：介绍了计算属性的概念和用法，它可以根据依赖的数据动态计算出一个新的值，并在模板中使用。

（7）监听器选项 watch：了解了如何使用 watch 选项来监测数据的变化，并在变化时执行相应的操作。

（8）模板引用：学习了如何在模板中使用模板引用来获取 DOM 元素或组件的引用，并在需要时进行操作。

（9）组件基础：探讨了 Vue.js 中组件的基本概念和使用方法。

虽然 Vue .js 3 中推出了组合式 API，但 Vue .js 2 的选项式 API 仍然在现有的 Vue.js 项目中非常常见。因此了解选项式 API 的知识对于理解和维护旧项目非常重要。在进阶篇中，我们将学习组合式 API，这是开发新项目时的首选方式。