

全部开发者教程 三

第11章: compiler 编译器 - 深入编辑器处理逻辑

01: 前言

02: 响应性数据的编辑器处理: 响应性数据的处理逻辑

03: 响应性数据的编辑器处理: AST 解析逻辑

04: 响应性数据的编辑器处理: JavaScript AST 转化逻辑

05: 响应性数据的编辑器处理: render 转化逻辑分析

06: 响应性数据的编辑器处理: generate 生成 render 函数

07: 响应性数据的编辑器处理: render 函数的执行处理

08: 多层级模板的编辑器处理: 多层级的处理逻辑

09: 基于编辑器的指令(v-xx)处理: 指令解析的整体逻辑



Sunday • 更新于 2022-10-19

◀ 上一节 08: 多层级模板... 10: 基于编辑器... 下一节 ▶

## 09: 基于编辑器的指令(v-xx)处理: 指令解析的整体逻辑

在 `vue` 中, 指令是一个非常重要的环节。 `vue` 的指令处理主要集中在 `compiler` 编辑器中。那么接下来我们就来看一下 `vue` 中的指令处理逻辑。

`vue` 中提供的指令非常多, 大家可以点击 [这里来查看所有的内置指令](#), 针对于那么多的指令, 我们不可能全部进行讲解实现逻辑, 所以我们在这里就以 `v-if` 为例, 来为大家讲解指令的解析与处理方案。

我们创建如下测试实例 `packages/vue/examples/imooc/compiler/compiler-directive.html`:

<> 代码块

```
1  <script>
2    const { compile, h, render } = Vue
3    // 创建 template
4    const template = `<div> hello world <h1 v-if="isShow">你好, 世界</h1> </div>`
5
6    // 生成 render 函数
7    const renderFn = compile(template)
8    console.log(renderFn.toString());
9    // 创建组件
10   const component = {
11     data() {
12       return {
13         isShow: false
14       }
15     },
16     render: renderFn
17   }
18
19   // 通过 h 函数, 生成 vnode
20   const vnode = h(component)
21
22   // 通过 render 函数渲染组件
23   render(vnode, document.querySelector('#app'))
24 </script>
```

查看生成的 `render` 函数:

<> 代码块

```
1  function render(_ctx, _cache) {
2    with (_ctx) {
3      const { openBlock: _openBlock, createElementBlock: _createElementBlock, createCommentVNode: _createCommentVNode } = true ? _sfc__vnodeTransformCache : {}
4
5      return (_openBlock(), _createElementBlock("div", null, [
6        _hoisted_1,
7        isShow
8        ? (_openBlock(), _createElementBlock("h1", _hoisted_2, "你好, 世界"))
9        : _createCommentVNode("v-if", true)
10      ]))
11    }
12  }
```

根据之前的经验和上面的代码可知:

### 索引目录

09: 基于编辑器的指令(v-xx)处理: 指令解析的整体逻辑



2. `isShow ? xx : xx`。这个三元表达式是渲染的关键。我们 `v-if` 本质上就是一个 `if` 判断，满足条件则渲染，不满足则不渲染。

那么明确好了对应的 `render` 逻辑之后，接下来我们就来看对应的 `ast` 和 `JavaScript AST`：

<> 代码块

```
1  {
2    "type": 0,
3    "children": [
4      {
5        "type": 1,
6        ...
7        "children": [
8          {
9            "type": 2,
10           ...
11         },
12         {
13           "type": 1, // NodeTypes.ELEMENT
14           "ns": 0,
15           "tag": "h1",
16           "tagType": 0,
17           "props": [
18             {
19               "type": 7, // NodeTypes.DIRECTIVE
20               "name": "if", // 指令名
21               // express: 表达式
22               "exp": {
23                 "type": 4, // NodeTypes.SIMPLE_EXPRESSION
24                 "content": "isShow", // 值
25                 "isStatic": false,
26                 "constType": 0,
27                 "loc": {...}
28               },
29               "modifiers": [],
30               "loc": {...}
31             }
32           ],
33           "isSelfClosing": false,
34           "children": [
35             {
36               "type": 2,
37               "content": "你好，世界",
38               "loc": {...}
39             }
40           ],
41           "loc": {... }
42         }
43       ],
44       ...
45     ]
46   },
47   ...
48 }
49
```

以上的 `AST`，我们进行了对应的简化，主要看备注部分。

由以上 `AST` 可知，针对于指令的处理，主要集中在 `props` 选项中，所以针对于 `AST` 而言，我们 **只需要额外增加 属性（`props`）的处理即可。**

接下来我们来看 `JavaScript AST`。

`JavaScript AST` 决定了最终的 `render` 渲染，它的处理更加复杂。我们之前创建过 `transformElement` 与 `transformText` 用来处理 `element` 和 `text` 的渲染，那么同样的道理，针对于指令的处理，我们也需要创建对应的 `transformXXX` 才可以进行处理。

如果以 `v-if` 为例，那么我们需要增加对应的 `vif.ts` 模块。

[意见反馈](#)

[收藏教程](#)

[标记书签](#)

<> 代码块

```
1  {
2    "type": 9,
3    // 分支处理
4    "branches": [
5      {
6        "type": 10, // NodeTypes.IF_BRANCH
7        "condition": {
8          "type": 4, // NodeTypes.SIMPLE_EXPRESSION
9          "content": "isShow",
10         "isStatic": false,
11         "loc": {}
12       },
13       "children": [
14         {
15           "type": 1,
16           "tag": "h1",
17           "tagType": 0,
18           "props": [],
19           "children": [{ "type": 2, "content": "你好，世界" }],
20           "codegenNode": {
21             "type": 13, // NodeTypes.VNODE_CALL
22             "tag": "\"h1\"",
23             "children": [{ "type": 2, "content": "你好，世界" }]
24           }
25         }
26       ]
27     },
28   ],
29   "codegenNode": {
30     "type": 19, // NodeTypes.JS_CONDITIONAL_EXPRESSION
31     "test": {
32       "type": 4,
33       "content": "isShow",
34       "isStatic": false,
35       "loc": {}
36     },
37     "consequent": {
38       "type": 13, // NodeTypes.VNODE_CALL
39       "tag": "\"h1\"",
40       "children": [{ "type": 2, "content": "你好，世界" }]
41     },
42     "alternate": {
43       "type": 14, // NodeTypes.JS_CALL_EXPRESSION
44       "callee": CREATE_COMMENT,
45       "loc": {},
46       "arguments": ["\"v-if\"", "true"]
47     },
48     "newline": true,
49     "loc": {}
50   }
51 }
52 ...
```

## 总结

到这里我们知道了，想要处理指令的编辑逻辑，那么 `AST` 和 `JavaScript AST`，我们都需要进行额外处理：

1. `AST`：额外增加 `props` 属性
2. `JavaScript AST`：额外增加 `branches` 属性

08: 多层次模板的编辑器处理: 多层次的处理... < 上一节 下一节 > 10: 基于编辑器的指令(v-x)处理: AST 解...

✍ 我要提出意见反馈

✍ 意见反馈

♥ 收藏教程

🔖 标记书签

