Sunday • 更新于 2022-10-19

上一节 01：前言 03：响应性数据... 下一节

02：响应性数据的编辑器处理：响应性数据的处理逻辑

那么首先我们先来看响应性数据的编辑器处理逻辑。他具体指的是什么呢？我们来看如下测试实例 `packages/vue/examples/compiler/compiler-rective.html` ：

代码块

```
<script>
  const { compile, h, render } = Vue
  // 创建 template
  const template = `<div> hello {{ msg }} </div>`

  // 生成 render 函数
  const renderFn = compile(template)

  // 创建组件
  const component = {
    data() {
      return {
        msg: 'world'
      }
    },
    render: renderFn
  }

  // 通过 h 函数，生成 vnode
  const vnode = h(component)

  // 通过 render 函数渲染组件
  render(vnode, document.querySelector('#app'))
</script>
```

在以上代码中，我们通过 `data` 声明了一个响应式数据，然后在 `tempalte` 中通过 `{{}}` 进行使用使用。从而得到了 `hello {{ msg }}` 这样一个表达式，这样的表达式我们把它叫做 **复合表达式**

我们可以在 `vue` 的源码的 `baseCompile` 方法中分别查看 `AST` 、 `JavaScript AST` 和 `render` 函数的值：

代码块

```
// AST
{
  "type": 0,
  "children": [
    {
      "type": 1,
      "tag": "div",
      "tagType": 0,
      "props": [],
      "children": [
        {
          "type": 2,
          "content": " hello ",
          "loc": {}
        },
        {
```

```
            "type": 4, // NodeTypes.SIMPLE_EXPRESSION
            "isStatic": false,
            "constType": 0,
            "content": "msg",
            "loc": {}
          },
          "loc": {}
        }
      ],
      "loc": {}
    }
  ],
  ...
}
```

代码块

```
// JavaScript AST
{
  "type": 0,
  "children": [
    {
      "type": 1,
      "ns": 0,
      "tag": "div",
      "tagType": 0,
      "props": [],
      "isSelfClosing": false,
      "children": [
        {
          "type": 8,
          "loc": {},
          "children": [
            {
              "type": 2,
              "content": " hello ",
              "loc": {}
            },
            " + ",
            {
              "type": 5, // NodeTypes.INTERPOLATION
              "content": {
                "type": 4, // NodeTypes.SIMPLE_EXPRESSION
                "isStatic": false,
                "constType": 0,
                "content": "msg",
                "loc": {}
              },
              "loc": {}
            }
          ]
        }
      ],
      "loc": {},
      "codegenNode": {...},
        "loc": {}
      }
    }
  ],
  "helpers": [Symbol("openBlock"), Symbol("createElementBlock"), Symbol("toDisplayString
  "codegenNode": {...},
    "loc": {}
  },
  "loc": {}
}
```

代码块

```
// context.code
```

```
return function render(_ctx, _cache) {
  with (_ctx) {
    const { toDisplayString: _toDisplayString, openBlock: _openBlock, createElementBlock

    return (_openBlock(), _createElementBlock("div", null, " hello " + _toDisplayString(
  }
}
```

由以上内容可以看出，当我们增加了复合表达式之后， `AST` 、 `JavaScript AST` 和 `render` 函数中多出了如下内容：

代码块

```
// AST
{
  "type": 5, // NodeTypes.INTERPOLATION
  "content": {
    "type": 4, // NodeTypes.SIMPLE_EXPRESSION
    "isStatic": false,
    "constType": 0,
    "content": "msg",
    "loc": {}
  }
}
```

代码块

```
// JavaScript AST
{
  "type": 8,
  "loc": {},
  "children": [
    {
      "type": 2,
      "content": " hello ",
      "loc": {}
    },
    " + ",
    {
      "type": 5, // NodeTypes.INTERPOLATION
      "content": {
        "type": 4, // NodeTypes.SIMPLE_EXPRESSION
        "isStatic": false,
        "constType": 0,
        "content": "msg",
        "loc": {}
      },
      "loc": {}
    }
  ]
}
```

代码块

```
// render
const _Vue = Vue

return function render(_ctx, _cache) {
+  with (_ctx) {
+    const { toDisplayString: _toDisplayString, openBlock: _openBlock, createElementBloc

+    return (_openBlock(), _createElementBlock("div", null, " hello " + _toDisplayString
+  }
}
```

那么当我们处理复合表达式的编译时，同样也是需要从差异入手，我们 **只需要填充对应的数据差异**，就可以完成最终 `render` 的生成。

我要提出意见反馈