Sunday • 更新于 2022-10-19

上一节 12：框架实现：… 14：源码阅读：… 下一节

13：扩展知识：render 函数的生成方案

当我们得到了 JavaScript AST 之后，下面我们就可以生成对应的 render 函数了。

那么我们如何根据 JavaScript AST 来生成对应的 render 函数呢？

我们先来看 vue 源码生成的 render：

1. 在 packages/compiler-core/src/compile.ts 中的 baseCompile 方法下，使用此代码（咱们自己生成的 JavaScript AST ）：

代码块

```
return generate(
  {
    type: 0,
    children: [
      {
        type: 1,
        tag: 'div',
        tagType: 0,
        props: [],
        children: [{ type: 2, content: ' hello world ' }],
        codegenNode: {
          type: 13,
          tag: '"div"',
          props: [],
          children: [{ type: 2, content: ' hello world ' }]
        }
      }
    ],
    loc: {},
    codegenNode: {
      type: 13,
      tag: '"div"',
      props: [],
      children: [{ type: 2, content: ' hello world ' }]
    },
    // 此处需要主动写入
    helpers: [CREATE_ELEMENT_VNODE],
    components: [],
    directives: [],
    imports: [],
    hoists: [],
    temps: [],
    cached: []
  },
  extend({}, options, {
    prefixIdentifiers
  })
)
```

代替原有的 generate 方法调用。

在 packages/compiler-core/src/codegen.ts 文件中的 generate 方法的最后位置，打印 context.code：

代码块

```
    console.log(context.code)
```

运行测试实例，可以得到如下打印：

代码块

```
const _Vue = Vue

return function render(_ctx, _cache) {
  with (_ctx) {
    const { createElementVNode: _createElementVNode } = _Vue



    return _createElementVNode("div", [], [" hello world "])
  }
}
```

该函数就是通过 `generate` 方法转化得到的 `render` 函数，在该 `render` 中存在一个 `with (_ctx)` 这个代码在我们最终期望得到的 `render` 函数中是不需要的。所以我们最终期望得到的 `render` 函数为：

代码块

```
const _Vue = Vue

return function render(_ctx, _cache) {
    const { createElementVNode: _createElementVNode } = _Vue

    return _createElementVNode("div", [], [" hello world "])
}
```

那么下面我们来分析一下上面这个函数的生成，即：生成方案。

函数的生成方案，分为三部分：

1. 函数本质上就是一段字符
2. 字符串的拼接方式
3. 字符串拼接的格式处理

函数本质上就是一段字符

函数本质上就是一段字符，所以我们可以把以上函数比较一个大的 **字符串** 。

那么想要生成这样的一个大字符串，本质上就是各个小的字符串的拼接。

例如，我们可以期望如下的拼接：

代码块

```
context.code = `
    const _Vue = Vue \n\n return function render(_ctx, _cache) { \n\n const { createElem
`
```

把以上字符串处理之后，我们就可以得到一样函数格式的字符：

代码块

```
context.code = `
    const _Vue = Vue

  return function render(_ctx, _cache) {
    const { createElementVNode: _createElementVNode } = _Vue
    return _createElementVNode("div", [], [" hello world "])
  }
`
```

当我们明确好了函数本身就是字符，这样的概念之后，那么接下来就是如何拼接这样的字符。

我们把上面的函数分成 4 个部分：

1. 函数的前置代码： `const _Vue = Vue`
2. 函数名： `function render`
3. 函数的参数： `_ctx, _cache`
4. 函数体：

代码块

```
    const { createElementVNode: _createElementVNode } = _Vue
    return _createElementVNode("div", [], [" hello world "])
```

我们只需要把以上的内容拼接到一起，那么就可以得到最终的目标结果。

那么为了完成对应的拼接，我们可以提供一个 `push` 函数：

代码块

```
function push (code) {
    context.code += code
}
```

以此来完成对应的拼接

关于字符串的格式

在去处理这样的一个字符串的过程中，我们不光需要处理拼接，还需要处理对应的格式问题，比如：

代码块

```
context.code = `
    const _Vue = Vue
  （换行）
  return function render(_ctx, _cache) {
  （缩进）const { createElementVNode: _createElementVNode } = _Vue
    return _createElementVNode("div", [], [" hello world "])
  }
`
```

对于字符串而言，我们知道换行可以通过 `\n` 来进行表示，缩进就是 空格的处理。

所以我们需要再提供对应的方法，来进行对应的处理，比如：

代码块

```
context.indentLevel = 0 // 表示缩进

// 换行
function newline(n: number) {
        newline(context.indentLevel)
}

// 缩进+换行
function indent(n: number) {
        newline(++context.indentLevel)
}

// 取消缩进 + 换行
function deindent(n: number) {
        newline(--context.indentLevel)
}

function newline(n: number) {
        context.code += '\n' + `  `.repeat(n)
}
```

我要提出意见反馈