☰ 逐行剖析 Vue.js 源码

1. 前言

经过前几篇文章，我们把用户所写的模板字符串先经过解析阶段解析生成对应的抽象语法树 AST ，接着再经过优化阶段将 AST 中的静态节点及静态根节点都打上标记，现在终于到了模板编译三大阶段的最后一个阶段了——代码生成阶段。所谓代码生成阶段，到底是要生成什么代码？答：要生成 render 函数字符串。

我们知道， Vue 实例在挂载的时候会调用其自身的 render 函数来生成实例上的 template 选项所对应的 VNode ，简单的来说就是 Vue 只要调用了 render 函数，就可以把模板转换成对应的虚拟 DOM 。那么 Vue 要想调用 render 函数，那必须要先有这个 render 函数，那这个 render 函数又是从哪来的呢？是用户手写的还是 Vue 自己生成的？答案是都有可能。我们知道，我们在日常开发中是可以在 Vue 组件选项中手写一个 render 选项，其值对应一个函数，那这个函数就是 render 函数，当用户手写了 render 函数时，那么 Vue 在挂载该组件的时候就会调用用户手写的这个 render 函数。那如果用户没有写呢？那这个时候 Vue 就要自己根据模板内容生成一个 render 函数供组件挂载的时候调用。而 Vue 自己根据模板内容生成 render 函数的过程就是本篇文章所要介绍的代码生成阶段。

现在我们知道了，所谓代码生成其实就是根据模板对应的抽象语法树 AST 生成一个函数，通过调用这个函数就可以得到模板对应的虚拟 DOM 。

2. 如何根据AST生成render函数

通过上文我们知道了，代码生成阶段主要的工作就是根据已有的 AST 生成对应的 render 函数供组件挂载时调用，组件只要调用的这个 render 函数就可以得到 AST 对应的虚拟 DOM 的 VNode 。那么如何根据 AST 生成 render 函数呢？这其中是怎样一个过程呢？接下来我们就来细细剖析一下。

假设现有如下模板：

```html
<div id="NLRX"><p>Hello {{name}}</p></div>
```

该模板经过解析并优化后对应的 AST 如下：

```js
ast = {
    'type': 1,
```

☰ 逐行剖析 Vue.js 源码

```
            {
                'name':'id',
                'value':'NLRX',
            }
        ],
        'attrsMap': {
          'id': 'NLRX',
        },
        'static':false,
        'parent': undefined,
        'plain': false,
        'children': [{
          'type': 1,
          'tag': 'p',
          'plain': false,
          'static':false,
          'children': [
            {
                'type': 2,
                'expression': '"Hello "+_s(name)',
                'text': 'Hello {{name}}',
                'static':false,
            }
          ]
        }]
    }
```

下面我们就来根据已有的这个 AST 来生成对应的 render 函数。生成 render 函数的过程其实就是一个递归的过程，从顶向下依次递归 AST 中的每一个节点，根据不同的 AST 节点类型创建不同的 VNode 类型。接下来我们就来对照已有的模板和 AST 实际演示一下生成 render 函数的过程。

1. 首先，根节点 div 是一个元素型 AST 节点，那么我们就要创建一个元素型 VNode ，我们把创建元素型 VNode 的方法叫做 _c(tagName,data,children) 。我们暂且不管 _c() 是什么，只需知道调用 _c() 就可以创建一个元素型 VNode 。那么就可以生成如下代码：

```js
_c('div',{attrs:{"id":"NLRX"}},[/*子节点列表*/])
```

2. 根节点 div 有子节点，那么我们进入子节点列表 children 里遍历子节点，发现子节点 p 也是元素型的，那就继续创建元素型 VNode 并将其放入上述代码中根节点的子节点列表中，如下：

☰ 逐行剖析 Vue.js 源码

3. 同理，继续遍历 `p` 节点的子节点，发现是一个文本型节点，那就创建一个文本型 `VNode` 并将其插入到 `p` 节点的子节点列表中，同理，创建文本型 `VNode` 我们调用 `_v()` 方法，如下：

```
_c('div',{attrs:{"id":"NLRX"}},[_c('p',{attrs:{}},[_v("Hello "+_s(nam
```

4. 到此，整个 `AST` 就遍历完毕了，我们将得到的代码再包装一下，如下：

```
`
with(this){
  reurn _c(
    'div',
    {
      attrs:{"id":"NLRX"},
    },
    [
      _c(
        'p',
        {
          attrs:{}
        },
        [
          _v("Hello "+_s(name))
        ]
      )
    ]
  )
}
`
```

5. 最后，我们将上面得到的这个函数字符串传递给 `createFunction` 函数（关于这个函数在后面会介绍到）， `createFunction` 函数会帮我们把得到的函数字符串转换成真正的函数，赋给组件中的 `render` 选项，从而就是 `render` 函数了。如下：

```
res.render = createFunction(compiled.render, fnGenErrors)

function createFunction (code, errors) {
```

☰ 逐行剖析 Vue.js 源码

```
    errors.push({ err, code })
    return noop
  }
}
```

以上就是根据一个简单的模板所对应的 AST 生成 render 函数的过程，理论过程我们已经了解了，那么在源码中实际是如何实现的呢？下面我们就回归源码分析其具体实现过程。

3. 回归源码

代码生成阶段的源码位于 src/compiler/codegen/index.js 中，源码虽然很长，但是逻辑不复杂，核心逻辑如下：

```js
export function generate (ast,option) {
  const state = new CodegenState(options)
  const code = ast ? genElement(ast, state) : '_c("div")'
  return {
    render: `with(this){return ${code}}`,
    staticRenderFns: state.staticRenderFns
  }
}
```

```js
const code = generate(ast, options)
```

调用 generate 函数并传入优化后得到的 ast ，在 generate 函数内部先判断 ast 是否为空，不为空则调用 genElement(ast, state) 函数创建 VNode ，为空则创建一个空的元素型 div 的 VNode 。然后将得到的结果用 with(this){return ${code}} 包裹返回。可以看出，真正起作用的是 genElement 函数，下面我们继续来看一下 genElement 函数内部是怎样的。

genElement 函数定义如下：

```js
export function genElement (el: ASTElement, state: CodegenState): string
  if (el.staticRoot && !el.staticProcessed) {
    return genStatic(el, state)
  } else if (el.once && !el.onceProcessed) {
    return genOnce(el, state)
```

☰ 逐行剖析 Vue.js 源码

```
    } else if (el.if && !el.ifProcessed) {
      return genIf(el, state)
    } else if (el.tag === 'template' && !el.slotTarget) {
      return genChildren(el, state) || 'void 0'
    } else if (el.tag === 'slot') {
      return genSlot(el, state)
    } else {
      // component or element
      let code
      if (el.component) {
        code = genComponent(el.component, el, state)
      } else {
        const data = el.plain ? undefined : genData(el, state)

        const children = el.inlineTemplate ? null : genChildren(el, state
        code = `_c('${el.tag}'${
          data ? `,${data}` : '' // data
        }${
          children ? `,${children}` : '' // children
        })`
      }
      // module transforms
      for (let i = 0; i < state.transforms.length; i++) {
        code = state.transforms[i](el, code)
      }
      return code
    }
  }
```

`genElement` 函数逻辑很清晰，就是根据当前 `AST` 元素节点属性的不同从而执行不同的代码生成函数。虽然元素节点属性的情况有很多种，但是最后真正创建出来的 `VNode` 无非就三种，分别是元素节点，文本节点，注释节点。接下来我们就着重分析一下如何生成这三种节点类型的 `render` 函数的。

3.1 元素节点

生成元素型节点的 `render` 函数代码如下：

```
const data = el.plain ? undefined : genData(el, state)

const children = el.inlineTemplate ? null : genChildren(el, state, true
```

☰ 逐行剖析 Vue.js 源码

```
children ? `,${children}` : '' // children
})`
```

生成元素节点的 `render` 函数就是生成一个 `_c()` 函数调用的字符串，上文提到了 `_c()` 函数接收三个参数，分别是节点的标签名 `tagName` ， 节点属性 `data` ， 节点的子节点列表 `children` 。那么我们只需将这三部分都填进去即可。

1. 获取节点属性data

首先判断 `plain` 属性是否为 `true` ，若为 `true` 则表示节点没有属性，将 `data` 赋值为 `undefined` ；如果不为 `true` 则调用 `genData` 函数获取节点属性 `data` 数据。`genData` 函数定义如下：

```js
export function genData (el: ASTElement, state: CodegenState): string
  let data = '{'
  const dirs = genDirectives(el, state)
  if (dirs) data += dirs + ','

    // key
    if (el.key) {
        data += `key:${el.key},`
    }
    // ref
    if (el.ref) {
        data += `ref:${el.ref},`
    }
    if (el.refInFor) {
        data += `refInFor:true,`
    }
    // pre
    if (el.pre) {
        data += `pre:true,`
    }
    // 篇幅所限，省略其他情况的判断
    data = data.replace(/,$/, '') + '}'
    return data
}
```

逐行剖析 Vue.js 源码

再加一个 `}` ，最终得到节点全部属性 `data` 。

2. 获取子节点列表children

获取子节点列表 `children` 其实就是遍历 `AST` 的 `children` 属性中的元素，然后根据元素属性的不同生成不同的 `VNode` 创建函数调用字符串，如下：

```js
export function genChildren (el):  {
    if (children.length) {
        return `[${children.map(c => genNode(c, state)).join(',')}]`
    }
}
function genNode (node: ASTNode, state: CodegenState): string {
  if (node.type === 1) {
    return genElement(node, state)
  } if (node.type === 3 && node.isComment) {
    return genComment(node)
  } else {
    return genText(node)
  }
}
```

3. 上面两步完成之后，生成 `_c()` 函数调用字符串，如下：

```js
code = `_c('${el.tag}'${
        data ? `,${data}` : '' // data
      }${
        children ? `,${children}` : '' // children
      })`
```

3.2 文本节点

文本型的 `VNode` 可以调用 `_v(text)` 函数来创建，所以生成文本节点的 `render` 函数就是生成一个 `_v(text)` 函数调用的字符串。 `_v()` 函数接收文本内容作为参数，如果文本是动态文本，则使用动态文本 `AST` 节点的 `expression` 属性，如果是纯静态文本，则使用 `text` 属性。其生成代码如下：

```
    ? text.expression // no need for () because already wrapped in _s()
    : transformSpecialNewlines(JSON.stringify(text.text))
  })`
}
```

3.3 注释节点

注释型的 `VNode` 可以调用 `_e(text)` 函数来创建，所以生成注释节点的 `render` 函数就是生成一个 `_e(text)` 函数调用的字符串。 `_e()` 函数接收注释内容作为参数，其生成代码如下：

```js
export function genComment (comment: ASTText): string {
  return `_e(${JSON.stringify(comment.text)})`
}
```

4. 总结

本篇文章介绍了模板编译三大阶段的最后一个阶段——代码生成阶段。

首先，介绍了为什么要有代码生成阶段以及代码生成阶段主要干什么。我们知道了，代码生成其实就是根据模板对应的抽象语法树 `AST` 生成一个函数供组件挂载时调用，通过调用这个函数就可以得到模板对应的虚拟 `DOM` 。

接着，我们通过一个简单的模板演示了把模板经过递归遍历最后生成 `render` 函数的过程。

最后，我们回归源码，通过分析源码了解了生成 `render` 函数的具体实现过程。

在 GitHub 上编辑此页

上次更新: 3/24/2020, 5:37:47 AM

← 优化阶段 总结 →