Sunday • 更新于 2022-10-19

上一节 15：框架实现：... 17：框架实现：... 下一节

16：框架实现：解析 JavaScript AST，拼接 render 函数

我们最终解析之后的目标函数如下：

代码块

```
const _Vue = Vue

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

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

依次，首先我们先生成 **除参数之外** 部分：

1. 在 `generate` 中：

代码块

```
/**
 * 根据 JavaScript AST 生成
 */
export function generate(ast) {
    // 生成上下文 context
    const context = createCodegenContext(ast)

    // 获取 code 拼接方法
    const { push, newline, indent, deindent } = context

    // 生成函数的前置代码：const _Vue = Vue
    genFunctionPreamble(context)

    // 创建方法名称
    const functionName = `render`
    // 创建方法参数
    const args = ['_ctx', '_cache']
    const signature = args.join(', ')

    // 利用方法名称和参数拼接函数声明
    push(`function ${functionName}(${signature}) {`)

    // 缩进 + 换行
    indent()

    // 明确使用到的方法。如：createVNode
    const hasHelpers = ast.helpers.length > 0
    if (hasHelpers) {
        push(`const { ${ast.helpers.map(aliasHelper).join(', ')} } = _Vue`)
        push(`\n`)
        newline()
    }

    // 最后拼接 return 的值
    newline()
    push(`return `)
```

```
39        ....
      }
```

2. 创建 `genFunctionPreamble` 方法：

代码块

```
1   /**
2    * 生成 "const _Vue = Vue\n\nreturn "
3    */
4   function genFunctionPreamble(context) {
5       const { push, newline, runtimeGlobalName } = context
6
7       const VueBinding = runtimeGlobalName
8       push(`const _Vue = ${VueBinding}\n`)
9
10      newline()
11      push(`return `)
12  }
```

3. 创建 `aliasHelper`：

代码块

```
1   const aliasHelper = (s: symbol) => `${helperNameMap[s]}: _${helperNameMap[s]}`
```

4. 运行此时的代码，我们应该可以得到这样的函数生成：

代码块

```
1   const _Vue = Vue
2
3   return function render(_ctx, _cache) {
4     const { createElementVNode: _createElementVNode } = _Vue
5
6
7     return
```

那么接下来我们就处理最后 `renturn` 函数的部分：

1. 补全 `generate` 中的代码：

代码块

```
1   /**
2    * 根据 JavaScript AST 生成
3    */
4   export function generate(ast) {
5       ...
6       // 处理 renturn 结果。如：_createElementVNode("div", [], [" hello world "])
7       if (ast.codegenNode) {
8           genNode(ast.codegenNode, context)
9       } else {
10          push(`null`)
11      }
12
13      // 收缩缩进 + 换行
14      deindent()
15      push(`}`)
16
17      return {
18          ast,
19          code: context.code
20      }
21  }
```

2. 创建 `genNode` 函数：

代码块

```
 */
function genNode(node, context) {
    switch (node.type) {
        case NodeTypes.VNODE_CALL:
            genVNodeCall(node, context)
            break
        case NodeTypes.TEXT:
            genText(node, context)
            break
    }
}
```

3. 创建 `genText` 函数：

代码块

```
/**
 * 处理 TEXT 节点
 */
function genText(node, context) {
    context.push(JSON.stringify(node.content), node)
}
```

4. 创建 `genVNodeCall` 函数：

代码块

```
/**
 * 处理 VNODE_CALL 节点
 */
function genVNodeCall(node, context) {
    const { push, helper } = context
    const { tag, props, children, patchFlag, dynamicProps, isComponent } = node

    // 返回 vnode 生成函数
    const callHelper = getVNodeHelper(context.inSSR, isComponent)
    push(helper(callHelper) + `(`, node)

    // 获取函数参数
    const args = genNullableArgs([tag, props, children, patchFlag, dynamicProps])

    // 处理参数的填充
    genNodeList(args, context)

    push(`)`)
}
```

5. 创建 `packages/compiler-core/src/utils.ts` 模块，添加 `getVNodeHelper` 方法：

代码块

```
/**
 * 返回 vnode 生成函数
 */
export function getVNodeHelper(ssr: boolean, isComponent: boolean) {
    return ssr || isComponent ? CREATE_VNODE : CREATE_ELEMENT_VNODE
}
```

6. 创建 `genNullableArgs` 函数：

代码块

```
/**
 * 处理 createXXXVnode 函数参数
 */
function genNullableArgs(args: any[]) {
    let i = args.length
    while (i--) {
        if (args[i] != null) break
    }
    return args.slice(0, i + 1).map((arg) => arg || `null`)
```

7. 创建 `genNodeList` 函数

代码块

```
/**
 * 处理参数的填充
 */
function genNodeList(nodes, context) {
    const { push, newline } = context
    for (let i = 0; i < nodes.length; i++) {
        const node = nodes[i]
        // 字符串直接 push 即可
        if (isString(node)) {
            push(node)
        }
        // 数组需要 push "[" "]"
        else if (isArray(node)) {
            genNodeListAsArray(node, context)
        }
        // 对象需要区分 node 节点类型，递归处理
        else {
            genNode(node, context)
        }
        if (i < nodes.length - 1) {
            push(', ')
        }
    }
}
```

8. 创建 `genNodeListAsArray` 函数：

代码块

```
function genNodeListAsArray(nodes, context) {
    context.push(`[`)
    genNodeList(nodes, context)
    context.push(`]`)
}
```

至此函数生成完成。接下来我们就来测试一下函数是否可用。

创建如下测试实例：

代码块

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

  // 生成 render 函数
  const { code } = compile(template)

  console.log(code);

  const renderFn = new Function(code)()


  // 创建组件
  const component = {
    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>
```

打印当前的 code 为：

代码块

```
const _Vue = Vue

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


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

由以上代码可知， `render` 函数使用到了 `createElementVNode` 方法，所以我们需要在 `runtime` 时，导出该方法：

1. 在 `packages/runtime-core/src/vnode.ts` 中，新增：

代码块

```
// createElementVNode 实际调用的是  createVNode
export { createVNode as createElementVNode }
```

2. 在 `packages/runtime-core/src/index.ts` 中，增加 `createElementVNode` 的导出：

代码块

```
export { ..., createElementVNode } from './vnode'
```

3. 在 `packages/vue/src/index.ts` 中，增加 `createElementVNode` 的导出

代码块

```
export {
    ...
    createElementVNode
} from '@vue/runtime-core'
```

此时，浏览器中，应该可以成功渲染。

15：框架实现：构建 CodegenContext 上下... ◂ 上一节　　下一节 ▸ 17：框架实现：新建 compat 模块，把 ren...

我要提出意见反馈