Sunday • 更新于 2022-10-19

上一节 13：扩展知识：... 15：框架实现：... 下一节

14：源码阅读：编译器第三步：生成 render 函数

我们知道生成 `render` 函数的代码，主要是 `packages/compiler-core/src/codegen.ts` 中的 `generate` 方法，所以我们可以直接在该方法中打断点，进入 `debugger` （**注意：此时我们使用的是 `vuex-next-mini` 生成 JavaScript AST**）：

1. 进入 `generate` 方法：
2. 执行 `const context = createCodegenContext(ast, options)` 得到 `context` 上下文：
    1. 进入 `createCodegenContext` 方法，对于 `context` 而言，我们现在是比较熟悉的了，知道它就是一个全局变量
    2. 观察该方法，可以发现 `context` 内部存在很多属性和方法，这些属性和方法很多，但是我们不需要全部关注，只需要关注如下内容即可：

代码块

```
    const context = {
        // render 函数代码字符串
        code: ``,
        // 运行时全局的变量名
        runtimeGlobalName: 'Vue',
        // 模板源
        source: ast.loc.source,
        // 缩进级别
        indentLevel: 0,
        // 需要触发的方法，关联 JavaScript AST 中的 helpers
        helper(key) {
            return `_${helperNameMap[key]}`
        },
        /**
         * 插入代码
         */
        push(code) {
            context.code += code
        },
        /**
         * 新的一行
         */
        newline() {
            newline(context.indentLevel)
        },
        /**
         * 控制缩进 + 换行
         */
        indent() {
            newline(++context.indentLevel)
        },
        /**
         * 控制缩进 + 换行
         */
        deindent() {
            newline(--context.indentLevel)
        }
    }
```

3. 这些代码相对而言，比较简单，我们在上一小节也提到过对应的作用，这里就不在赘述了。

3. 执行完成该方法之后，我们可以得到一个 `context.code` 目前值为 “”
4. 接下来的代码执行，就是不断往 `context.code` 填充内容的过程
5. 代码执行 `genFunctionPreamble(ast, preambleContext)`：

    1. 进入 `genFunctionPreamble` 方法
    2. 执行 `if (ast.helpers.length > 0)` 满足条件

        1. 执行 `push(` const _Vue = ${VueBinding}\n `)`
        2. 当前的 `VueBinding = Vue`，所以以上等同于 `push(const _Vue = Vue\n)`
        3. 此时，**context.code = "const _Vue = Vue\n"**

    3. 执行 `newline()`，此时，**context.code = "const _Vue = Vue\n\n"**
    4. 执行 `push(` return `)`，此时，**context.code = "const _Vue = Vue\n\nreturn"**

6. `genFunctionPreamble` 执行完成，此时，**context.code = "const _Vue = Vue\n\nreturn"**
7. 代码继续执行，生成 `functionName` 和 `args`

8. 执行 `push(` function $functionName$({signature}) { `)`，此时，**context.code = ""“const _Vue = Vue\n\nreturn function render(_ctx, _cache) {””"**
9. 执行 `indent()`，此时，**context.code = "const _Vue = Vue\n\nreturn function render(_ctx, _cache) {\n "**
10. 执行 `push(` with (_ctx) { `)` 。

    1. 此时，**context.code = "const _Vue = Vue\n\nreturn function render(_ctx, _cache) {\n with (_ctx) {"**

11. 执行 `indent()` 。此时，**context.code = "const _Vue = Vue\n\nreturn function render(_ctx, _cache) {\n with (_ctx) { \n"**
12. 执行:

    代码块

    ```
    1    push(`const { ${ast.helpers.map(aliasHelper).join(', ')} } = _Vue`)
    2    push(`\n`)
    3    newline()
    ```

13. 此时，

    代码块

    ```
    1    context.code =  "const _Vue = Vue\n\nreturn function render(_ctx, _cache) {\n  with
    ```

14. 执行 `push(` return `)`
15. 此时:

    代码块

    ```
    1    context.code =  "const _Vue = Vue\n\nreturn function render(_ctx, _cache) {\n  with
    ```

16. 那么到此为止，对于 `code` 而言，就只剩下最后一块内容，也就是：

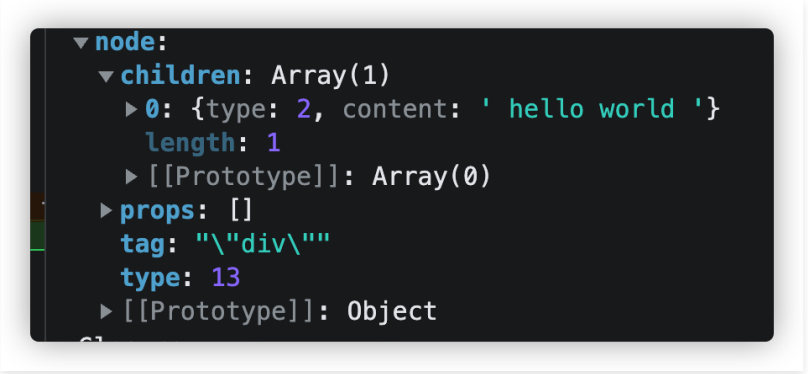
    代码块

    ```
    1    _createElementVNode("div", [], [" hello world "])
    ```

17. 而这里，也是整个 `generate` 最复杂的一块逻辑

18. 这块逻辑由 `genNode(ast.codegenNode, context)` 开始，我们进入到 `genNode` 方法

    1. 进入 `genNode` 方法，目前的参数 `node` 为：



    2. 代码执行 `switch`：

        1. 当前的 `type` 为 `13`，对应 `case NodeTypes.VNODE_CALL`
        2. 所以触发 `genVNodeCall` 方法

            1. 进入 `genVNodeCall` 方法
            2. 代码执行 `const callHelper: symbol = xxx`，这里的 `isBlock = undefined`，所以会触发 `getVNodeHelper(context.inSSR, isComponent)` 方法：

                1. 进入 `getVNodeHelper` 方法：
                2. 该方法的内部执行非常简单：

                ```
                代码块
                1    return ssr || isComponent ? CREATE_VNODE : CREATE_ELEMENT_VNODE
                ```

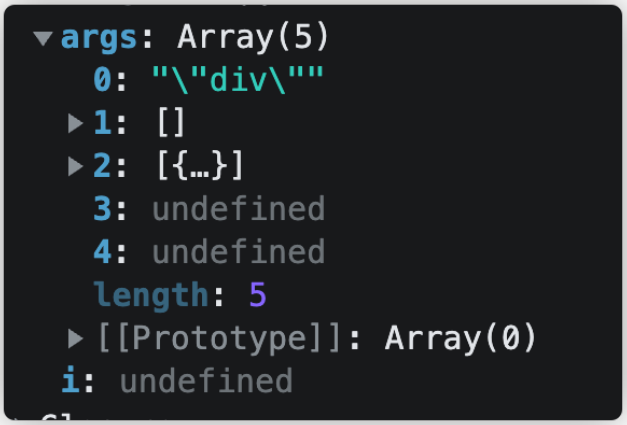
                3. 返回了两个 `Symbol`，分别对应 `createVNode` 和 `createElementVNode` 方法
                4. 此处返回 `CREATE_ELEMENT_VNODE`

            3. 执行 `push(helper(callHelper) + ( , node)`
            4. 此时：

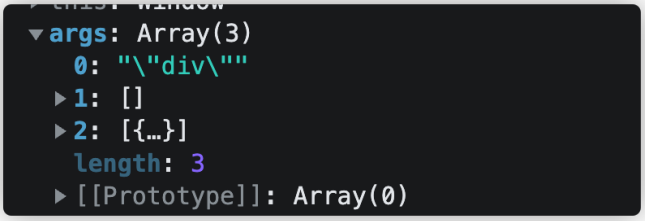
            ```
            代码块
            1    context.code =  "const _Vue = Vue\n\nreturn function render(_ctx, _cache)
            ```

            5. 接下来我们就需要为 方法填充参数：
            6. 执行 `const args = genNullableArgs(...)`

                1. 进入 `genNullableArgs` 方法，此时的 `arg` 参数为：



2. 执行 `for` 循环，最终返回值为：



7. 代码执行 `genNodeList(args, context)` ， **处理参数的** `push`

    1. 执行 `for` 循环，循环会被触发 `3` 次：

        1. 第一次触发： `node = 'div'`

            1. 直接执行 `push(node)`
            2. 执行 `push(', ')`

        2. 第二次触发： `node =[]`

            1. 执行 `genNodeListAsArray(node, context)`

                1. 执行 `context.push( [ )`
                2. 执行 `context.push( ] )`

            2. 跳出方法，执行 `push(', ')`

        3. 第三次触发：

            1. 执行 `genNodeListAsArray(node, context)`

                1. 执行 `context.push( [ )`
                2. 执行 `genNodeList(nodes, context, multilines)`

                    1. 通常触发 `for` 循环，此时： `node` 的值为：

2. 执行 `genNode(node, context)`

1.

13：扩展知识：render 函数的生成方案 ◂ 上一节　　下一节 ▸ 15：框架实现：构建 CodegenContext 上下...

我要提出意见反馈