Sunday • 更新于 2022-10-19

上一节 08：扩展知识：... 10：框架实现：... 下一节

09：源码阅读：编译器第二步：转化 AST，得到 JavaScript AST 对象

这一小节，我们就来看下对应的 `transform` 逻辑，我们创建如下测试实例：

代码块

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

  // 生成 render 函数
  const renderFn = compile(template)
</script>
```

进入 `baseCompile` 方法，触发 `debugger`：

1. 进入 `baseCompile` 方法：
2. 执行 `transform`，**注意：** 此时 `tansform` 方法传递了两个参数：
    1. `ast`：这是我们在 `parse` 时生成的 `AST` 对象
    2. `options`：这是一个配置对象，里面包含了上一小节说的 `nodeTransforms`
3. 触发 `transform` 方法，**该方法即为转化 `JavaScript AST` 的核心方法**：
    1. 进入 `transform` 方法
    2. 执行 `createTransformContext`，该方法主要为生成 `context` 上下文对象
        1. 进入 `createTransformContext` 方法
        2. 该方法内部的代码很多，但是整体逻辑比较简单，核心就是创建了一个对象 `context`，然后执行了 `return`
        3. 在生成的对象中，我们只需要关注如下几个核心属性即可：

代码块

```
{
    /**
     * AST 根节点
     */
    root
    /**
     * 每次转化时记录的父节点
     */
    parent: ParentNode | null
    /**
     * 每次转化时记录的子节点索引
     */
    childIndex: number
    /**
     * 当前处理的节点
     */
    currentNode
```

```
19          * 协助创建 JavaScript AST 属性 helpers，该属性是一个数组，值为 Symbol(方法名
20          */
21         helpers: Map<symbol, number>
22         helper<T extends symbol>(name: T): T
23         /**
24          * 转化方法集合
25          */
26         nodeTransforms: any[]
27     }
```

3. 得到 `context` 上下文对象
4. 之后，**执行** `traverseNode` 方法，该方法为转化的 **核心方法**：
    1. 进入 `traverseNode` ，此时参数为：
        1. `node` ： `ast` 对象
        2. `context` ： 上下文对象
    2. 执行 `context.currentNode = node` ，标记当前处理的节点。**此时** `currentNode` **为最外层的** `root` **节点**
    3. 执行 `const { nodeTransforms } = context` ，获取 `nodeTransforms` 数组，该数组中封装了所有的节点转化方法
    4. 执行 `const exitFns = []` 和 `for` 循环，该循环的主要作用是：**往** `exitFns` **数组中依次放入** `nodeTransform` **转化方法**：
        1. 进入该循环
        2. 执行 `const onExit = nodeTransforms[i](node, context)` ，获取转化方法
            1. 进入 `nodeTransforms[i](node, context)` 中进行查看
            2. 可以发现虽然执行了很多不同的 `transformXXX` 方法，但是这些方法都有一个共同点就是： `return` **了一个方法。**即： `transformXXX` **方法是一个闭包函数，对外返回了一个待执行的方法**
            3. 返回的方法现在未执行，将来会在 `exitFns[i]()` 的时候被触发。
            4. 所以现在我们不着急查看
    5. 循环执行完成。 `exitFns` 中将放入所有的 `transformXXX` 方法，此时对应的 `node` **为** `root` ，即：**最顶层对象。**当前 `exitFns` 中的值为：

```
1     [
2         // 转化 element
3       postTransformElement(),
4       // 转化 text
5       transformText()
6     ]
```

    6. 执行 `switch (node.type)` ，进入 `switch` ：
        1. 执行 `traverseChildren(node, context)` ，该方法会 **循环处理所有的子节点**
            1. 进入 `traverseChildren` 方法
            2. 可以看到方法本身的逻辑比较简单，会遍历所有的子节点，以触发 `traverseNode`
            3. 我们尝试再次进入 `traverseNode` 来进行查看
            4. 再次进入 `traverseNode` ，此时参数为：
                1. `node` ： `root` 下 `children` 的第一个节点，即：** `type = ELEMENT` 的 `div` **
                2. `context` ： 上下文对象
            5. 再次执行 `for` 循环逻辑，填充 `exitFns` ，与上次不同的是此时对应的 `node` **为** `ty`

6. 再次触发 `switch`，执行 `traverseChildren(node, context)` 方法，**循环处理子节点**

7. **依次迭代**，当所有的迭代执行完成之后，代码继续往下执行

8. 代码通过 `switch`，继续往下执行时，**将是从最底层（`text` 节点）开始处理的**

9. 此时将执行 **依次退出** 逻辑，在依次退出时，会依次触发 `exitFns` 中保存的方法：

   代码块

   ```
   let i = exitFns.length
   while (i--) {
       exitFns[i]()
   }
   ```

   这里大家需要注意：**这是一个 `i--` 的逻辑**，这样的逻辑意味着 **保存的方法将从后往前执行**

10. 之前的时候我们说过，`exitFns[i]()` 中保存着所有的 `transformXXX` 方法，每次 `exitFns[i]()` 执行都意味着一个 `transformXXX` 触发，即：**一个节点被转化**

那么至此，我们的 `traverseNode` 方法的讲解就算是完成了，下面我们就需要进入 `transformXXX` 函数的处理，我们这里主要使用到了两个 `transformXXX` 方法，分别为：

1. `packages/compiler-core/src/transforms/transformElement.ts` 中的 `transformElement` 方法
2. `packages/compiler-core/src/transforms/transformText.ts` 中的 `transformText` 方法

那么下面我们依次来看这两个方法，这两个方法会被多次触发，所以我们可以直接在 `exitFns[i]()` 中增加断点：

1. 执行 `context.currentNode = node`，**明确当前的执行节点**

2. 第一次触发

   1. 第一次触发 `transformElement` 方法，其实我们这里触发的验证来说应该是 `return` 的 `postTransformElement` 函数
   2. `node = context.currentNode!`，利用我们的 `context` 的上下文对象，获取到当前的 `node` 节点，此时的 `node` 节点是：

      ```
      ▼node:
        content: " hello world "
      ▶ loc: {start: {…}, end: {…}, source: ' hello world '}
        type: 2
      ▶ [[Prototype]]: Object
      ```

   3. 该节点为 **最底层** 的节点，满足我们之前所说的深度优先
   4. 代码触发 `if`，不符合条件，直接 `renturn`

3. 第二次触发

   1. 第二次触发 `transformText` 方法

   2. 执行 `const children = node.children`，此时的 `children` 为：

      ```
      ▼children: Array(1)
        ▼0:
          content: " hello world "
        ▶ loc: {start: {…}, end: {…}, source: ' hello world '}
          type: 2
        ▶ [[Prototype]]: Object
          length: 1
        ▶ [[Prototype]]: Array(0)
      ```

   3. 针对于 `transformText` 方法而言，代码非常多，但是并不复杂，当前我们的代码没有办法满足它的运行场景，所以我们直接描述一下它的逻辑：

```
/**
 * 方法的作用：将相邻的文本节点和表达式合并为一个表达式。
 *
 * 例如：
 * <div>hello {{ msg }}</div>
 * 上述模板包含两个节点：
 * 1. hello：TEXT 文本节点
 * 2. {{ msg }}：INTERPOLATION 表达式节点
 * 这两个节点在生成 render 函数时，需要被合并： 'hello' + _toDisplayString(_ctx.msg)
 * 那么在合并时就要多出来这个 + 加号。
 * 例如：
 * children:[
 *  { TEXT 文本节点 },
 *  " + ",
 *  { INTERPOLATION 表达式节点 }
 * ]
 */
```

4. 因为我们当前没有 `{{}}`，所以我们无法观察后续执行，后续代码直接跳过

4. 第三次触发

    1. 第三次触发 `transformElement` 方法，此时的 `node` 节点为：

```
▼ node:
  ▶ children: [{…}]
    codegenNode: undefined
    isSelfClosing: false
  ▶ loc: {start: {…}, end: {…}, source: '<div> hello world </div>'}
    ns: 0
  ▶ props: []
    tag: "div"
    tagType: 0
    type: 1
```

    2. 满足条件，触发逻辑
    3. 对于该方法而言，内部的代码非常多，但是处理我们无需关注，比如： `props`、 `children` ...
    4. 我们直接关注 `node.codegenNode = createVNodeCall(...)` 的逻辑

        1. 进入 `createVNodeCall` 方法
        2. 执行 `context.helper(getVNodeHelper(context.inSSR, isComponent))`：

            1. 在这里就利用到了我们生成 `context` 的时候，创建的 `helper` 方法和 `getVNodeHelper` 方法，我们分别进入这两个方法来看一下

                1. 进入 `getVNodeHelper` 方法

                    1. 内部的逻辑非常简单，只是一个三元表达式，直接返回了一个 `CREATE_ELEMENT_VNODE` 的常量，对应的值为 `Symbol(createElementVNode)`
                    2. 该常量在生成 `render` 函数时代表了 `createElementVNode` 方法：

                2. 进入 `helper` 方法

                    1. 该方法也非常简单，只是把刚才的 `CREATE_ELEMENT_VNODE` 这个常量放入到了 `helpers` 对象中。
                    2. 针对于 `helpers` 对象：

                        1. `key`：函数名
                        2. `value`：索引

        3. 最后 `return` 一个对象

    5. 该对象即为 `codegenNode` 对象

那么至此，我们看到了两个主要的 `transformXXX` 方法的执行，后面还会存在多次的执行，我们就不在一个一个去看了。

当所有的 `transformXXX` 执行完成之后，意味着整个 `traverseNode` 全部执行完成。 `traverseNode` 执行完成标记着此时：

1. `root` **的** `children` **中将包含有** `codegenNode` **对象**
2. 所有的 **文本节点和表达式** 也都完成了合并

此时所有的 `children` 都有了 `codegenNode` 对象，但是对于最外层的 `root` 还不存在 `codegenNode` ，所有接下来我们要处理最外层的 `codegenNode`

那么此时我们可以再回到 `transform` 方法中，继续往下执行：

1. 重新回到 `transform` 方法
2. 执行 `createRootCodegen(root, context)` 方法：
    1. 进入 `createRootCodegen`
    2. 执行 `const { children } = root` 拿到 `children`
    3. 我们当前 **只有一个根节点**，所以 `children.length = 0`
    4. 执行 `if (isSingleElementRoot(root, child) && child.codegenNode)` ，确认当前只存在一个根节点
    5. **拿到第一个子节点的** `codegenNode` ，使其为 `root` 的 `codegenNode`
3. 至此 `root` 的 `codegenNode` 存在值，值为 **第一个子节点的** `codegenNode`
4. 最后执行：

    代码块

    ```
    root.helpers = [...context.helpers.keys()]
      root.components = [...context.components]
      root.directives = [...context.directives]
      root.imports = context.imports
      root.hoists = context.hoists
      root.temps = context.temps
      root.cached = context.cached
    ```

    完成各中值的初始化即可

由以上代码可知：

1. 整个 `transform` 的逻辑可以大致分为两部分：
    1. 深度优先排序，通过 `traverseNode` 方法，完成排序逻辑
    2. 通过保存在 `nodeTransforms` 中的 `transformXXX` 方法，针对不同的节点，完成不同的处理
2. 期间创建的 `context` 上下文，承担了一个全局单例的作用

08：扩展知识：AST 到 JavaScript AST 的转... ◂ 上一节　　下一节 ▸ 10：框架实现：转化 JavaScript AST，构建...

我要提出意见反馈

企业服务　网站地图　网站首页　关于我们　联系我们　讲师招募　帮助中心　意见反馈　代码托管

意见反馈　收藏教程　标记书签