全部开发者教程

第十三章：compiler 编译器 - 编译时核心设计原则

01：前言

02：模板编译的核心流程

03：抽象语法树 - AST

04：AST 转化为 JavaScript AST，获取 codegenNode

05：JavaScript AST 生成 render 函数代码

06：总结

第十四章：compiler 编译器 - 构建 compile 编译器

01：前言

02：扩展知识：JavaScript与有限自动状态机

03：扩展知识：扫描 tokens 构建 AST 结构的方案

04：源码阅读：编译器第一步：依据模板，生成 AST 抽象语法树

Sunday • 更新于 2022-10-19

上一节 03：扩展知识：... 05：框架实现：... 下一节

04：源码阅读：编译器第一步：依据模板，生成 AST 抽象语法树

那么这一小节我们就来看一下 vue 中生成 AST 的代码。该部分代码全部被放入到了 packages/compiler-core/src/parse.ts 中，从这个文件可以看出，整体 parse 的逻辑非常复杂，整体文件有 1173 行代码。

所以我们再去看这一块逻辑的时候，同样会按照之前的方式，只去关注当前业务下的逻辑，而忽略其他逻辑，依次来降低整体的复杂度。

通过 packages/compiler-core/src/compile.ts 中的 baseCompile 方法可以知道，整个 parse 的过程是从 baseParse 开始的，所以我们可以直接从这个方法开始进行 debugger 。

测试实例为：

代码块

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

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

当前的 template 对应的目标极简 AST 为（这意味着我们将不再关注其他的属性生成）：

代码块

```
const ast = {
  "type": 0,
  "children": [
    {
      "type": 1,
      "tag": "div",
      "tagType": 0,
      // 属性，目前我们没有做任何处理。但是需要添加上，否则，生成的 ats 放到 vue 源码中会抛出错
      "props": [],
      "children": [{ "type": 2, "content": " hello world " }]
    }
  ],
  // loc：位置，这个属性并不影响渲染，但是它必须存在，否则会报错。所以我们给了他一个 {}
  "loc": {}
}

```

模板解析的 token 流程为（以 hello world 为例）：

代码块

```
1. <div
2. >
3. hello world
4. </div
5. >
```

索引目录

04：源码阅读：编

1. 进入 `baseParse` 方法
2. 执行 `createParserContext`，**生成** `context` **上下文对象**

    1. 进入 `createParserContext` 方法
    2. 该方法中返回了一个 `ParserContext` 类型的对象：

        1. `ParserContext` 是一个解析器上下文对象，里面包含了非常多的解析器属性
        2. 具体可查看 `packages/compiler-core/src/parse.ts` 中 第 `92` 行

    3. 该对象比较复杂，我们只需要关注 `source`（模板源代码） 这一个属性即可

3. 此时 `context.source = "<div> hello world </div>"`
4. 执行 `getCursor(context)` 方法，该方法主要获取 `loc` （即：location 位置），与我们的极简 `AST` 无关，**无需关注**
5. **执行** `parseChildren` **方法（解析子节点）**，这个方法 **非常重要**，是生成 `AST` 的核心方法：

    1. 进入 `parseChildren` 方法
    2. 执行 `const nodes: TemplateChildNode[] = []`，创建 `nodes` 变量，这个 `nodes` 就是生成的 `AST` 中的 `children`
    3. 执行 `while` 循环，循环解析模板数据：

        1. 循环的判断条件为 `!isEnd(context, mode, ancestors)` ，我们进入到 `isEnd` 方法进行查看

            1. 执行 `const s = context.source`，获取 `s`，此时 `s = <div> hello world </div>`
            2. 不符合 `isEnd` 的条件，返回 `false`，**进入循环**

        2. 执行 `const s = context.source` ，此时的 `s = <div> hello world </div>`
        3. 执行 `let node`，声明 `node`，这个 `node` 就是 `children` 中的元素
        4. 执行 `if (mode === TextModes.DATA || mode === TextModes.RCDATA) {...}` 和 `else if (mode === TextModes.DATA && s[0] === '<')` ，因为当前的 `s = <div> hello world </div>` ，所以 **满足条件。**表示为：**标签开始**

            1. 执行 `else if (/[a-z]/i.test(s[1]))`，**满足条件。**表示为：**以** `<` **开始，后面跟** `a-z` **表示，这是一个标签的开始**
            2. 执行 `node = parseElement(context, ancestors)` 方法。即：`parseElement` **的返回值为** `node`，我们知道 `node` 为 `children` 下的元素，所以说：`parseElement` **即为解析** `element` **，生成** `children` **下元素的方法**

                1. 进入 `parseElement` 方法，**开始解析** `element`，此时 `context.source = <div> hello world </div>`

                    1. 整个 `parseElement` 的解析分为三步：

                        1. 开始标签：例如 `<div>`
                        2. 子节点：例如 `hello world`
                        3. 结束标签：例如 `</div>`

                2. 首先执行 **开始标签** `<div>` 的解析：

                    1. 执行 `const element = parseTag(context, TagType.Start, parent)` 方法，`parseTag` 表示为 **解析标签**

                        1. 整个 `parseTag` 方法解析标签共分为两步：

                            1. 标签开始：例如：`<div`
                            2. 标签结束：例如：`>`

2. 进入 `parseTag` 方法，该方法为 **解析标签** 的方法，主要做了 **四件** 事情：

    1. 首先处理 **标签开始**

        1. 代码执行： `const match = /^<\/?([a-z][^\t\r\n\f />]*)/i.exec(context.source)!`
        2. 代码执行： `const tag = match[1]` 。利用 `match` 这个正则，拿到 `tag` 标签名，此时标签名为 `tag = div`
        3. 执行 `advanceBy(context, match[0].length)` 方法，此处的 **advanceBy** 方法 **非常重要。**

            1. 该方法的的作用，主要为：**解析模板**
            2. 针对于 `<div>hello world</div>` 而言，一共会被解析 `5` 次，解析的顺序为：

                1.

03：扩展知识：扫描 tokens 构建 AST 结构... ◂ 上一节　　下一节 ▸ 05：框架实现：构建 parse 方法，生成 cont...