Sunday • 更新于 2022-10-19

上一节 05：框架实现：... 07：框架实现：... 下一节

06：框架实现：构建有限自动状态机解析模板，扫描 token 生成 AST 结构

接下来我们通过 `parseChildren` 方法处理所有的子节点，整个处理的过程分为两大块：

1. 构建有限自动状态机解析模板
2. 扫描 token 生成 AST 结构

接下来我们来进行实现：

1. 创建 `parseChildren` 方法：

代码块

```
/**
 * 解析子节点
 * @param context 上下文
 * @param mode 文本模型
 * @param ancestors 祖先节点
 * @returns
 */
function parseChildren(context: ParserContext, ancestors) {
    // 存放所有 node节点数据的数组
    const nodes = []

    /**
     * 循环解析所有 node 节点，可以理解为对 token 的处理。
     * 例如：<div>hello world</div>，此时的处理顺序为：
     * 1. <div
     * 2. >
     * 3. hello world
     * 4. </
     * 5. div>
     */
    while (!isEnd(context, ancestors)) {
        /**
         * 模板源
         */
        const s = context.source
        // 定义 node 节点
        let node

        if (startsWith(s, '{{')) {
        }
        // < 意味着一个标签的开始
        else if (s[0] === '<') {
            // 以 < 开始，后面跟a-z 表示，这是一个标签的开始
            if (/[a-z]/i.test(s[1])) {
                // 此时要处理 Element
                node = parseElement(context, ancestors)
            }
        }

        // node 不存在意味着上面的两个 if 都没有进入，那么我们就认为此时的 token 为文本节点
        if (!node) {
```

```
44
45          pushNode(nodes, node)
46      }
47
48      return nodes
49  }
```

2. 以上代码中涉及到了 个方法：

1. `isEnd`：判断是否为结束节点
2. `startsWith`：判断是否以指定文本开头
3. `pushNode`：为 `array` 执行 `push` 方法
4. **复杂：** `parseElement`：解析 `element`
5. **复杂：** `parseText`：解析 `text`

3. 我们先实现前三个简单方法：

4. 创建 `startsWith` 方法：

代码块

```
1   /**
2    * 是否以指定文本开头
3    */
4   function startsWith(source: string, searchString: string): boolean {
5       return source.startsWith(searchString)
6   }
```

5. 创建 `isEnd` 方法：

代码块

```
1   /**
2    * 判断是否为结束节点
3    */
4   function isEnd(context: ParserContext, ancestors): boolean {
5       const s = context.source
6
7       // 解析是否为结束标签
8       if (startsWith(s, '</')) {
9           for (let i = ancestors.length - 1; i >= 0; --i) {
10              if (startsWithEndTagOpen(s, ancestors[i].tag)) {
11                  return true
12              }
13          }
14      }
15      return !s
16  }
```

6. `isEnd` 方法中使用了 `startsWithEndTagOpen` 方法，所以我们要实现它：

代码块

```
1   /**
2    * 判断当前是否为《标签结束的开始》。比如 </div> 就是 div 标签结束的开始
3    * @param source 模板。例如：</div>
4    * @param tag 标签。例如：div
5    * @returns
6    */
7   function startsWithEndTagOpen(source: string, tag: string): boolean {
8       return (
9           startsWith(source, '</') &&
10          source.slice(2, 2 + tag.length).toLowerCase() === tag.toLowerCase() &&
11          /[\t\r\n\f />]/.test(source[2 + tag.length] || '>')
12      )
13  }
```

7. 创建 `pushNode` 方法：

```
/**
 * nodes.push(node)
 */
function pushNode(nodes, node): void {
    nodes.push(node)
}
```

至此三个简单的方法都被构建完成。

接下来我们来处理 `parseElement`，在处理的过程中，我们需要使用到 `NodeTypes` 和 `ElementTypes` 这两个 `enum` 对象，所以我们需要先构建它们（直接复制即可）：

1. 创建 `packages/compiler-core/src/ast.ts` 模块：

代码块

```
/**
 * 节点类型（我们这里复制了所有的节点类型，但是我们实际上只用到了极少的部分）
 */
export const enum NodeTypes {
    ROOT,
    ELEMENT,
    TEXT,
    COMMENT,
    SIMPLE_EXPRESSION,
    INTERPOLATION,
    ATTRIBUTE,
    DIRECTIVE,
    // containers
    COMPOUND_EXPRESSION,
    IF,
    IF_BRANCH,
    FOR,
    TEXT_CALL,
    // codegen
    VNODE_CALL,
    JS_CALL_EXPRESSION,
    JS_OBJECT_EXPRESSION,
    JS_PROPERTY,
    JS_ARRAY_EXPRESSION,
    JS_FUNCTION_EXPRESSION,
    JS_CONDITIONAL_EXPRESSION,
    JS_CACHE_EXPRESSION,

    // ssr codegen
    JS_BLOCK_STATEMENT,
    JS_TEMPLATE_LITERAL,
    JS_IF_STATEMENT,
    JS_ASSIGNMENT_EXPRESSION,
    JS_SEQUENCE_EXPRESSION,
    JS_RETURN_STATEMENT
}

/**
 * Element 标签类型
 */
export const enum ElementTypes {
    /**
     * element，例如：<div>
     */
    ELEMENT,
    /**
     * 组件
     */
    COMPONENT,
    /**
     * 插槽
     */
    SLOT,
    /**
```

```
56          */
57         TEMPLATE
58     }
```

下面就可以构建 `parseElement` 方法了，该方法的作用主要为了解析 `Element` 元素：

1. 创建 `parseElement`：

代码块

```
1   /**
2    * 解析 Element 元素。例如：<div>
3    */
4   function parseElement(context: ParserContext, ancestors) {
5       // -- 先处理开始标签 --
6       const element = parseTag(context, TagType.Start)
7
8       //  -- 处理子节点 --
9       ancestors.push(element)
10      // 递归触发 parseChildren
11      const children = parseChildren(context, ancestors)
12      ancestors.pop()
13      // 为子节点赋值
14      element.children = children
15
16      //  -- 最后处理结束标签 --
17      if (startsWithEndTagOpen(context.source, element.tag)) {
18          parseTag(context, TagType.End)
19      }
20
21      // 整个标签处理完成
22      return element
23  }
```

2. 构建 `TagType enum`：

代码块

```
1   /**
2    * 标签类型，包含：开始和结束
3    */
4   const enum TagType {
5       Start,
6       End
7   }
```

3. 处理开始标签，构建 `parseTag`：

代码块

```
1   /**
2    * 解析标签
3    */
4   function parseTag(context: any, type: TagType): any {
5       // -- 处理标签开始部分 --
6
7       // 通过正则获取标签名
8       const match: any = /^<\/?([a-z][^\r\n\t\f />]*)/i.exec(context.source)
9       // 标签名字
10      const tag = match[1]
11
12      // 对模板进行解析处理
13      advanceBy(context, match[0].length)
14
15      // -- 处理标签结束部分 --
16
17      // 判断是否为自关闭标签，例如 <img />
18      let isSelfClosing = startsWith(context.source, '/>')
19      // 《继续》对模板进行解析处理，是自动标签则处理两个字符 /> ，不是则处理一个字符 >
20      advanceBy(context, isSelfClosing ? 2 : 1)
```

```
    let tagType = ElementTypes.ELEMENT

    return {
        type: NodeTypes.ELEMENT,
        tag,
        tagType,
        // 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
        props: []
    }
}
```

4. 解析标签的过程，其实就是一个自动状态机不断读取的过程，我们需要构建 `advanceBy` 方法，来标记进入下一步：

代码块

```
/**
 * 前进一步。多次调用，每次调用都会处理一部分的模板内容
 * 以 <div>hello world</div> 为例
 * 1. <div
 * 2. >
 * 3. hello world
 * 4. </div
 * 5. >
 */
function advanceBy(context: ParserContext, numberOfCharacters: number): void {
    // template 模板源
    const { source } = context
    // 去除开始部分的无效数据
    context.source = source.slice(numberOfCharacters)
}
```

至此 `parseElement` 构建完成。此处的代码虽然不多，但是逻辑非常复杂。在解析的过程中，会再次触发 `parseChildren`，这次触发表示触发 文本解析，所以下面我们要处理 `parseText` 方法。

1. 创建 `parseText` 方法，解析文本：

代码块

```
/**
 * 解析文本。
 */
function parseText(context: ParserContext) {
    /**
     * 定义普通文本结束的标记
     * 例如：hello world </div>，那么文本结束的标记就为 <
     * PS：这也意味着如果你渲染了一个 <div> hell<o </div> 的标签，那么你将得到一个错误
     */
    const endTokens = ['<', '{{']
    // 计算普通文本结束的位置
    let endIndex = context.source.length

    // 计算精准的 endIndex，计算的逻辑为：从 context.source 中分别获取 '<', '{{' 的下标，
    for (let i = 0; i < endTokens.length; i++) {
        const index = context.source.indexOf(endTokens[i], 1)
        if (index !== -1 && endIndex > index) {
            endIndex = index
        }
    }

    // 获取处理的文本内容
    const content = parseTextData(context, endIndex)

    return {
        type: NodeTypes.TEXT,
        content
    }
}
```

2. 解析文本的过程需要获取到文本内容，此时我们需要构建 `parseTextData` 方法：

代码块

```
/**
 * 从指定位置（length）获取给定长度的文本数据。
 */
function parseTextData(context: ParserContext, length: number): string {
    // 获取指定的文本数据
    const rawText = context.source.slice(0, length)
    // 《继续》对模板进行解析处理
    advanceBy(context, length)
    // 返回获取到的文本
    return rawText
}
```

最后在 `baseParse` 中触发 `parseChildren` 方法：

代码块

```
/**
 * 基础的 parse 方法，生成 AST
 * @param content tempalte 模板
 * @returns
 */
export function baseParse(content: string) {
    // 创建 parser 对象，未解析器的上下文对象
    const context = createParserContext(content)
    const children = parseChildren(context, [])
    console.log(children)
    return {}
}
```

此时运行测试实例，应该可以打印出如下内容：

代码块

```
[
  {
    "type": 1,
    "tag": "div",
    "tagType": 0,
    "props": [],
    "children": [{ "type": 2, "content": " hello world " }]
  }
]
```

05：框架实现：构建 parse 方法，生成 cont... 上一节　　下一节 07：框架实现：生成 `AST`，构建测试

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