> 本文由 简悦 SimpRead 转码， 原文地址 kaiwu.lagou.com

这一节课我们依然要**解析 template 生成 AST 背后的实现原理**，上节课，我们知道了 baseParse 主要就做三件事情：**创建解析上下文**，**解析子节点**，**创建 AST 根节点。**

我们讲到了解析子节点，主要有四种情况，分别是注释节点的解析、插值的解析、普通文本的解析，以及元素节点的解析，这节课我们就到了最后的元素节点。

解析子节点

- 元素节点的解析

最后，我们来看元素节点的解析过程，它会解析模板中的标签节点，举个例子：

```
<div>


<hello :msg="msg"></hello>


</div>
```

相对于前面三种类型的解析过程，元素节点的解析过程应该是最复杂的了，即当前代码 s 是以 < 开头，并且后面跟着字母，说明它是一个标签的开头，则走到元素节点的解析处理逻辑，我们来看 parseElement 的实现：

```
function parseElement(context, ancestors) {


const wasInPre = context.inPre


const wasInVPre = context.inVPre


const parent = last(ancestors)


const element = parseTag(context, 0 , parent)


const isPreBoundary = context.inPre && !wasInPre


const isVPreBoundary = context.inVPre && !wasInVPre


if (element.isSelfClosing || context.options.isVoidTag(element.tag)) {


return element


  }
```

```
  ancestors.push(element)

const mode = context.options.getTextMode(element, parent)

const children = parseChildren(context, mode, ancestors)

  ancestors.pop()

  element.children = children

if (startsWithEndTagOpen(context.source, element.tag)) {

    parseTag(context, 1 , parent)

  }

else {

    emitError(context, 24 , 0, element.loc.start);

if (context.source.length === 0 && element.tag.toLowerCase() === 'script') {

const first = children[0];

if (first && startsWith(first.loc.source, '<!--')) {

        emitError(context, 8 )

      }

    }

  }

  element.loc = getSelection(context, element.loc.start)

if (isPreBoundary) {
```

```
    context.inPre = false

  }

if (isVPreBoundary) {

    context.inVPre = false

  }

return element

}
```

可以看到，这个过程中 parseElement 主要做了三件事情：解析开始标签，解析子节点，解析闭合标签。

首先，我们来看解析开始标签的过程。主要通过 parseTag 方法来解析并创建一个标签节点，来看它的实现原理：

```
function parseTag(context, type, parent) {

const start = getCursor(context)

const match = /^<\/?([a-z][^\t\r\n\f />]*)/i.exec(context.source);

const tag = match[1];

const ns = context.options.getNamespace(tag, parent);

  advanceBy(context, match[0].length);

  advanceSpaces(context);

const cursor = getCursor(context);

const currentSource = context.source;

  let props = parseAttributes(context, type);
```

```
if (context.options.isPreTag(tag)) {

    context.inPre = true;

  }

if (!context.inVPre &&

    props.some(p => p.type === 7  && p.name === 'pre')) {

    context.inVPre = true;

    extend(context, cursor);

    context.source = currentSource;

    props = parseAttributes(context, type).filter(p => p.name !== 'v-pre');

  }

  let isSelfClosing = false;

if (context.source.length === 0) {

    emitError(context, 9 );

  }

else {

    isSelfClosing = startsWith(context.source, '/>');

if (type === 1  && isSelfClosing) {

      emitError(context, 4 );

    }
```

```
    advanceBy(context, isSelfClosing ? 2 : 1);

  }

  let tagType = 0 ;

const options = context.options;

if (!context.inVPre && !options.isCustomElement(tag)) {

const hasVIs = props.some(p => p.type === 7  && p.name === 'is');

if (options.isNativeTag && !hasVIs) {

if (!options.isNativeTag(tag))

        tagType = 1 ;

    }

else if (hasVIs ||

      isCoreComponent(tag) ||

      (options.isBuiltInComponent && options.isBuiltInComponent(tag)) ||

      /^[A-Z]/.test(tag) ||

      tag === 'component') {

      tagType = 1 ;

    }

if (tag === 'slot') {

      tagType = 2 ;

    }
```

```
else if (tag === 'template' &&

      props.some(p => {

return (p.type === 7  && isSpecialTemplateDirective(p.name));

      })) {

      tagType = 3 ;

    }

  }

return {

    type: 1 ,

    ns,

    tag,

    tagType,

    props,

    isSelfClosing,

    children: [],

    loc: getSelection(context, start),

    codegenNode: undefined

  };

}
```

parseTag 首先匹配标签文本结束的位置，并前进代码到标签文本后面的空白字符后，然后解析标签中的属性，比如 class、style 和指令等，parseAttributes 函数的实现我就不多说了，感兴趣的同学可以自己去看，它最终会解析生成一个 props 的数组，并前进代码到属性后。

接着去检查是不是一个 pre 标签，如果是则设置 context.inPre 为 true；再去检查属性中有没有 v-pre 指令，如果有则设置 context.inVPre 为 true，并重置上下文 context 和重新解析属性；接下来再去判断是不是一个自闭和标签，并前进代码到闭合标签后；最后判断标签类型，是组件、插槽还是模板。

parseTag 最终返回的值就是一个描述标签节点的对象，其中 type 表示它是一个标签节点，tag 表示标签名，tagType 表示标签的类型，content 表示文本的内容，isSelfClosing 表示是否是一个闭合标签，loc 表示文本的代码开头和结束的位置信息，children 是标签的子节点数组，会先初始化为空。

解析完开始标签后，再回到 parseElement，接下来第二步就是解析子节点，它把解析好的 element 节点添加到 ancestors 数组中，然后执行 parseChildren 去解析子节点，并传入 ancestors。

如果有嵌套的标签，那么就会递归执行 parseElement，可以看到，在 parseElement 的一开始，我们能获取 ancestors 数组的最后一个值拿到父元素的标签节点，这个就是我们在执行 parseChildren 前添加到数组尾部的。

解析完子节点后，我们再把 element 从 ancestors 中弹出，然后把 children 数组添加到 element.children 中，同时也把代码前进到子节点的末尾。

最后，就是解析结束标签，并前进代码到结束标签后，然后更新标签节点的代码位置。parseElement 最终返回的值就是这样一个标签节点 element。

其实 HTML 的嵌套结构的解析过程，就是一个递归解析元素节点的过程，为了维护父子关系，当需要解析子节点时，我们就把当前节点入栈，子节点解析完毕后，我们就把当前节点出栈，因此 ancestors 的设计就是一个栈的数据结构，整个过程是一个不断入栈和出栈的过程。

通过不断地递归解析，我们就可以完整地解析整个模板，并且标签类型的 AST 节点会保持对子节点数组的引用，这样就构成了一个树形的数据结构，所以整个解析过程构造出的 AST 节点数组就能很好地映射整个模板的 DOM 结构。

空白字符管理

在前面的解析过程中，有些时候我们会遇到空白字符的情况，比如前面的例子：

```
<div>


<hello :msg="msg"></hello>


</div>
```

div 标签到下一行会有一个换行符，hello 标签前面也有空白字符，这些空白字符在解析的过程中会被当作文本节点解析处理。但这些空白节点显然是没有什么意义的，所以我们需要移除这些节点，减少后续对这些没用意义的节点的处理，以提高编译效率。

我们先来看一下空白字符管理相关逻辑代码：

```
function parseChildren(context, mode, ancestors) {


const parent = last(ancestors)
```

```
const ns = parent ? parent.ns : 0

const nodes = []

  let removedWhitespace = false

if (mode !== 2 ) {

if (!context.inPre) {

for (let i = 0; i < nodes.length; i++) {

const node = nodes[i]

if (node.type === 2 ) {

if (!/[^\t\r\n\f ]/.test(node.content)) {

const prev = nodes[i - 1]

const next = nodes[i + 1]

if (!prev ||

               !next ||

               prev.type === 3  ||

               next.type === 3  ||

               (prev.type === 1  &&

                 next.type === 1  &&

                 /[\r\n]/.test(node.content))) {

               removedWhitespace = true
```

```
              nodes[i] = null

            }

else {

              node.content = ' '

            }

          }

else {

            node.content = node.content.replace(/[\t\r\n\f ]+/g, ' ')

          }

        }

else if (!(process.env.NODE_ENV !== 'production') && node.type === 3 ) {

          removedWhitespace = true

          nodes[i] = null

        }

      }

    }

else if (parent && context.options.isPreTag(parent.tag)) {

const first = nodes[0]

if (first && first.type === 2 ) {
```

```
        first.content = first.content.replace(/^\r?\n/, '')

      }

    }

  }

return removedWhitespace ? nodes.filter(Boolean) : nodes

}
```

这段代码逻辑很简单，主要就是遍历 nodes，拿到每一个 AST 节点，判断是否为一个文本节点，如果是则判断它是不是空白字符；如果是则进一步判断空白字符是开头或还是结尾节点，或者空白字符与注释节点相连，或者空白字符在两个元素之间并包含换行符，如果满足上述这些情况，这些空白字符节点都应该被移除。

此外，不满足这三种情况的空白字符都会被压缩成一个空格，非空文本中间的空白字符也会被压缩成一个空格，在生产环境下注释节点也会被移除。

在 parseChildren 函数的最后，会过滤掉这些被标记清除的节点并返回过滤后的 AST 节点数组。

创建 AST 根节点

子节点解析完毕，baseParse 过程就剩最后一步创建 AST 根节点了，我们来看一下 createRoot 的实现：

```
function createRoot(children, loc = locStub) {

return {

    type: 0 ,

    children,

    helpers: [],

    components: [],

    directives: [],

    hoists: [],
```

```
    imports: [],

    cached: 0,

    temps: 0,

    codegenNode: undefined,

    loc

  }

}
```

createRoot 的实现非常简单，它就是返回一个 JavaScript 对象，作为 AST 根节点。其中 type 表示它是一个根节点类型，children 是我们前面解析的子节点数组。除此之外，这个根节点还添加了其它的属性，当前我们并不需要搞清楚每一个属性代表的含义，这些属性我们在分析后续的处理流程中会介绍。

总结

好的，到这里我们这一节的学习也要结束啦，通过这节课的学习，你应该掌握 Vue.js 编译过程的第一步，即**把 template 解析生成 AST 对象**，整个解析过程是一个自顶向下的分析过程，也就是从代码开始，通过语法分析，找到对应的解析处理逻辑，创建 AST 节点，处理的过程中也在不断前进代码，更新解析上下文，最终根据生成的 AST 节点数组创建 AST 根节点。

最后，给你留一道思考题目，在 parseTag 的过程中，如果解析的属性有 v-pre 标签，为什么要回到之前的 context，重新解析一次？欢迎你在留言区与我分享。

> **本节课的相关代码在源代码中的位置如下：**
> packages/compiler-core/src/parse.ts
> packages/compiler-core/src/ast.ts