☰ 逐行剖析 Vue.js 源码

1. 前言

上篇文章中我们说到，在模板解析阶段主线函数 `parse` 中，根据要解析的内容不同会调用不同的解析器，

而在三个不同的解析器中最主要的当属 `HTML` 解析器，为什么这么说呢？因为 `HTML` 解析器主要负责解析出模板字符串中有哪些内容，然后根据不同的内容才能调用其他的解析器以及做相应的处理。那么本篇文章就来介绍一下 `HTML` 解析器是如何解析出模板字符串中包含的不同的内容的。

2. HTML解析器内部运行流程

在源码中， `HTML` 解析器就是 `parseHTML` 函数，在模板解析主线函数 `parse` 中调用了该函数，并传入两个参数，代码如下：

```js
// 代码位置：/src/complier/parser/index.js

/**
 * Convert HTML string to AST.
 * 将HTML模板字符串转化为AST
 */
export function parse(template, options) {
   // ...
  parseHTML(template, {
    warn,
    expectHTML: options.expectHTML,
    isUnaryTag: options.isUnaryTag,
    canBeLeftOpenTag: options.canBeLeftOpenTag,
    shouldDecodeNewlines: options.shouldDecodeNewlines,
    shouldDecodeNewlinesForHref: options.shouldDecodeNewlinesForHref,
    shouldKeepComment: options.comments,
    // 当解析到开始标签时，调用该函数
    start (tag, attrs, unary) {

    },
    // 当解析到结束标签时，调用该函数
    end () {

    },
```

逐行剖析 Vue.js 源码

```
    },
    // 当解析到注释时，调用该函数
    comment (text) {

    }
  })
  return root
}
```

从代码中我们可以看到，调用 `parseHTML` 函数时为其传入的两个参数分别是：

- template:待转换的模板字符串；
- options:转换时所需的选项；

第一个参数是待转换的模板字符串，无需多言；重点看第二个参数，第二个参数提供了一些解析 `HTML` 模板时的一些参数，同时还定义了4个钩子函数。这4个钩子函数有什么作用呢？我们说了模板编译阶段主线函数 `parse` 会将 `HTML` 模板字符串转化成 `AST` ，而 `parseHTML` 是用来解析模板字符串的，把模板字符串中不同的内容出来之后，那么谁来把提取出来的内容生成对应的 `AST` 呢？答案就是这4个钩子函数。

把这4个钩子函数作为参数传给解析器 `parseHTML` ，当解析器解析出不同的内容时调用不同的钩子函数从而生成不同的 `AST` 。

- 当解析到开始标签时调用 `start` 函数生成元素类型的 `AST` 节点，代码如下；

```js
// 当解析到标签的开始位置时，触发start
start (tag, attrs, unary) {
    let element = createASTElement(tag, attrs, currentParent)
}

export function createASTElement (tag,attrs,parent) {
  return {
    type: 1,
    tag,
    attrsList: attrs,
    attrsMap: makeAttrsMap(attrs),
    parent,
    children: []
  }
}
```

createASTElement 函数来创建元素类型的 AST 节点

- 当解析到结束标签时调用 end 函数；
- 当解析到文本时调用 chars 函数生成文本类型的 AST 节点；

```js
// 当解析到标签的文本时，触发chars
chars (text) {
    if(text是带变量的动态文本){
    let element = {
      type: 2,
      expression: res.expression,
      tokens: res.tokens,
      text
    }
  } else {
    let element = {
      type: 3,
      text
    }
  }
}
```

当解析到标签的文本时，触发 chars 钩子函数，在该钩子函数内部，首先会判断文本是不是一个带变量的动态文本，如“hello ”。如果是动态文本，则创建动态文本类型的 AST 节点；如果不是动态文本，则创建纯静态文本类型的 AST 节点。

- 当解析到注释时调用 comment 函数生成注释类型的 AST 节点；

```js
// 当解析到标签的注释时，触发comment
comment (text: string) {
  let element = {
    type: 3,
    text,
    isComment: true
  }
}
```

当解析到标签的注释时，触发 comment 钩子函数，该钩子函数会创建一个注释类型的 AST 节点。

3. 如何解析不同的内容

要从模板字符串中解析出不同的内容，那首先要知道模板字符串中都会包含哪些内容。那么通常我们所写的模板字符串中都会包含哪些内容呢？经过整理，通常模板内会包含如下内容：

- 文本，例如“难凉热血”
- HTML注释，例如<!-- 我是注释 -->
- 条件注释，例如<!-- [if !IE]> -->我是注释<!--< ![endif] -->
- DOCTYPE，例如<!DOCTYPE html>
- 开始标签，例如
- 结束标签，例如

这几种内容都有其各自独有的特点，也就是说我们要根据不同内容所具有的不同的的特点通过编写不同的正则表达式将这些内容从模板字符串中一一解析出来，然后再把不同的内容做不同的处理。

下面，我们就来分别看一下 `HTML` 解析器是如何从模板字符串中将以上不同种类的内容进行解析出来。

3.1 解析HTML注释

解析注释比较简单，我们知道 `HTML` 注释是以 `<!--` 开头，以 `-->` 结尾，这两者中间的内容就是注释内容，那么我们只需用正则判断待解析的模板字符串 `html` 是否以 `<!--` 开头，若是，那就继续向后寻找 `-->` ，如果找到了，OK，注释就被解析出来了。代码如下：

```js
const comment = /^<!\--/
if (comment.test(html)) {
  // 若为注释，则继续查找是否存在'-->'
  const commentEnd = html.indexOf('-->')

  if (commentEnd >= 0) {
    // 若存在 '-->',继续判断options中是否保留注释
    if (options.shouldKeepComment) {
      // 若保留注释，则把注释截取出来传给options.comment，创建注释类型的AST节点
      options.comment(html.substring(4, commentEnd))
    }
    // 若不保留注释，则将游标移动到'-->'之后，继续向后解析
    advance(commentEnd + 3)
```

☰ 逐行剖析 Vue.js 源码

```
}
```

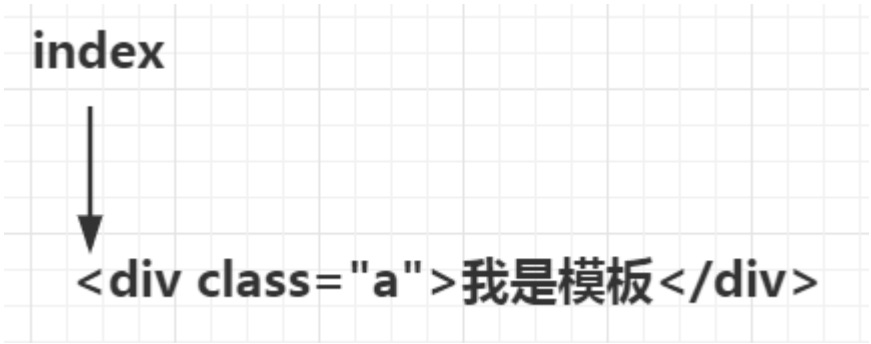
在上面代码中，如果模板字符串 `html` 符合注释开始的正则，那么就继续向后查找是否存在 `-->` ，若存在，则把 `html` 从第4位（"<!--"长度为4）开始截取，直到 `-->` 处，截取得到的内容就是注释的真实内容，然后调用4个钩子函数中的 `comment` 函数，将真实的注释内容传进去，创建注释类型的 `AST` 节点。

上面代码中有一处值得注意的地方，那就是我们平常在模板中可以在 `<template></template>` 标签上配置 `comments` 选项来决定在渲染模板时是否保留注释，对应到上面代码中就是 `options.shouldKeepComment` ,如果用户配置了 `comments` 选项为 `true` ，则 `shouldKeepComment` 为 `true` ，则创建注释类型的 `AST` 节点，如不保留注释，则将游标移动到'-->'之后，继续向后解析。

`advance` 函数是用来移动解析游标的，解析完一部分就把游标向后移动一部分，确保不会重复解析，其代码如下：

```js
function advance (n) {
  index += n    // index为解析游标
  html = html.substring(n)
}
```

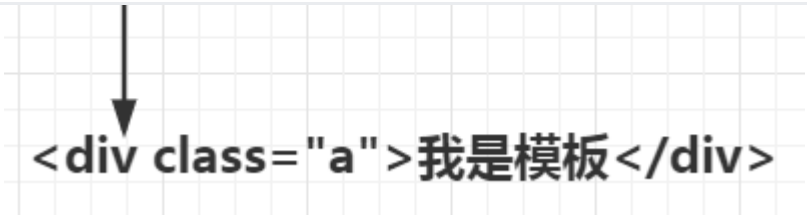
为了更加直观地说明 `advance` 的作用，请看下图：



调用 `advance` 函数：

```js
advance(3)
```

得到结果：



从图中可以看到，解析游标 `index` 最开始在模板字符串的位置0处，当调用了 `advance(3)` 之后，解析游标到了位置3处，每次解析完一段内容就将游标向后移动一段，接着再从解析游标往后解析，这样就保证了解析过的内容不会被重复解析。

3.2 解析条件注释

解析条件注释也比较简单，其原理跟解析注释相同，都是先用正则判断是否是以条件注释特有的开头标识开始，然后寻找其特有的结束标识，若找到，则说明是条件注释，将其截取出来即可，由于条件注释不存在于真正的 `DOM` 树中，所以不需要调用钩子函数创建 `AST` 节点。代码如下：

```js
// 解析是否是条件注释
const conditionalComment = /^<!\[/
if (conditionalComment.test(html)) {
  // 若为条件注释，则继续查找是否存在']>'
  const conditionalEnd = html.indexOf(']>')

  if (conditionalEnd >= 0) {
    // 若存在 ']>',则从原本的html字符串中把条件注释截掉，
    // 把剩下的内容重新赋给html，继续向后匹配
    advance(conditionalEnd + 2)
    continue
  }
}
```

3.3 解析DOCTYPE

解析 `DOCTYPE` 的原理同解析条件注释完全相同，此处不再赘述，代码如下：

```js
const doctype = /^<!DOCTYPE [^>]+>/i
// 解析是否是DOCTYPE
const doctypeMatch = html.match(doctype)
if (doctypeMatch) {
```

逐行剖析 Vue.js 源码

3.4 解析开始标签

相较于前三种内容的解析，解析开始标签会稍微复杂一点，但是万变不离其宗，它的原理还是相通的，都是使用正则去匹配提取。

首先使用开始标签的正则去匹配模板字符串，看模板字符串是否具有开始标签的特征，如下：

```js
/**
 * 匹配开始标签的正则
 */
const ncname = '[a-zA-Z_][\\w\\-\\.]*'
const qnameCapture = `((?:${ncname}\\:)?${ncname})`
const startTagOpen = new RegExp(`^<${qnameCapture}`)

const start = html.match(startTagOpen)
if (start) {
  const match = {
    tagName: start[1],
    attrs: [],
    start: index
  }
}

// 以开始标签开始的模板：
'<div></div>'.match(startTagOpen)  => ['<div','div',index:0,input:'<div>
// 以结束标签开始的模板：
'</div><div></div>'.match(startTagOpen) => null
// 以文本开始的模板：
'我是文本</p>'.match(startTagOpen) => null
```

在上面代码中，我们用不同类型的内容去匹配开始标签的正则，发现只有 `<div></div>` 的字符串可以正确匹配，并且返回一个数组。

在前文中我们说到，当解析到开始标签时，会调用4个钩子函数中的 `start` 函数，而 `start` 函数需要传递3个参数，分别是标签名 `tag` 、标签属性 `attrs` 、标签是否自闭合 `unary` 。标签名通过正则匹配的结果就可以拿到，即上面代码中的 `start[1]` ，而标签属性 `attrs` 以及标签是否自闭合 `unary` 需要进一步解析。

逐行剖析 Vue.js 源码

我们知道，标签属性一般是写在开始标签的标签名之后的，如下：

```html
<div class="a" id="b"></div>
```

另外，我们在上面匹配是否是开始标签的正则中已经可以拿到开始标签的标签名，即上面代码中的 `start[0]` ，那么我们可以将这一部分先从模板字符串中截掉，则剩下的部分如下：

```html
 class="a" id="b"></div>
```

那么我们只需用剩下的这部分去匹配标签属性的正则，就可以将标签属性提取出来了，如下：

```js
const attribute = /^\s*([^\s"'<>\/=]+)(?:\s*(=)\s*(?:"([^"]*)"+|'([^'
let html = 'class="a" id="b"></div>'
let attr = html.match(attribute)
console.log(attr)
// ["class="a"", "class", "=", "a", undefined, undefined, index: 0, i
```

可以看到，第一个标签属性 `class="a"` 已经被拿到了。另外，标签属性有可能有多个也有可能没有，如果没有的话那好办，匹配标签属性的正则就会匹配失败，标签属性就为空数组；而如果标签属性有多个的话，那就需要循环匹配了，匹配出第一个标签属性后，就把该属性截掉，用剩下的字符串继续匹配，直到不再满足正则为止，代码如下：

```js
const attribute = /^\s*([^\s"'<>\/=]+)(?:\s*(=)\s*(?:"([^"]*)"+|'([^'
const startTagClose = /^\s*(\/?)>/
const match = {
 tagName: start[1],
 attrs: [],
 start: index
}
while (!(end = html.match(startTagClose)) && (attr = html.match(attri
 advance(attr[0].length)
 match.attrs.push(attr)
}
```

就进入循环，继续提取，直到把所有标签属性都提取完毕。

所谓不符合开始标签的结束特征是指当前剩下的字符串不是以开始标签结束符开头的，我们知道一个开始标签的结束符有可能是一个 `>` （非自闭合标签），也有可能是 `/>` （自闭合标签），如果剩下的字符串（如 `></div>` ）以开始标签的结束符开头，那么就表示标签属性已经被提取完毕了。

2. 解析标签是否是自闭合

在 `HTML` 中，有自闭合标签（如 `<img src=""/>` ）也有非自闭合标签（如 `<div></div>` ），这两种类型的标签在创建 `AST` 节点是处理方式是有区别的，所以我们需要解析出当前标签是否是自闭合标签。

解析的方式很简单，我们知道，经过标签属性提取之后，那么剩下的字符串无非就两种，如下：

```html
<!--非自闭合标签-->
></div>
```

或

```html
<!--自闭合标签-->
/>
```

所以我们可以用剩下的字符串去匹配开始标签结束符正则，如下：

```js
const startTagClose = /^\s*(\/?)>/
let end = html.match(startTagClose)
'></div>'.match(startTagClose) // [">", "", index: 0, input: "></div>
'/>'.match(startTagClose) // ["/>", "/", index: 0, input: "/><div></d
```

可以看到，非自闭合标签匹配结果中的 `end[1]` 为 `""` ，而自闭合标签匹配结果中的 `end[1]` 为 `"/"` 。所以根据匹配结果的 `end[1]` 是否是 `""` 我们即可判断出当前标签是否为自闭合标签，源码如下：

```js
const startTagClose = /^\s*(\/?)>/
let end = html.match(startTagClose)
```

逐行剖析 Vue.js 源码

```
    advance(end[0].length)
    match.end = index
    return match
  }
```

经过以上两步，开始标签就已经解析完毕了，完整源码如下：

```js
const ncname = '[a-zA-Z_][\\w\\-\\.]*'
const qnameCapture = `((?:${ncname}\\:)?${ncname})`
const startTagOpen = new RegExp(`^<${qnameCapture}`)
const startTagClose = /^\s*(\/?)>/


function parseStartTag () {
  const start = html.match(startTagOpen)
  // '<div></div>'.match(startTagOpen)  => ['<div','div',index:0,input:
  if (start) {
    const match = {
      tagName: start[1],
      attrs: [],
      start: index
    }
    advance(start[0].length)
    let end, attr
    /**
     * <div a=1 b=2 c=3></div>
     * 从<div之后到开始标签的结束符号'>'之前，一直匹配属性attrs
     * 所有属性匹配完之后，html字符串还剩下
     * 自闭合标签剩下：'/>'
     * 非自闭合标签剩下：'></div>'
     */
    while (!(end = html.match(startTagClose)) && (attr = html.match(att
      advance(attr[0].length)
      match.attrs.push(attr)
    }

    /**
     * 这里判断了该标签是否为自闭合标签
     * 自闭合标签如:<input type='text' />
     * 非自闭合标签如:<div></div>
     * '></div>'.match(startTagClose) => [">", "", index: 0, input: "><
     * '/><div></div>'.match(startTagClose) => ["/>", "/", index: 0, in
     * 因此，我们可以通过end[1]是否是"/"来判断该标签是否是自闭合标签
     */
```

逐行剖析 Vue.js 源码

```
        advance(end[0].length)
        match.end = index
        return match
      }
    }
  }
```

通过源码可以看到，调用 `parseStartTag` 函数，如果模板字符串符合开始标签的特征，则解析开始标签，并将解析结果返回，如果不符合开始标签的特征，则返回 `undefined` 。

解析完毕后，就可以用解析得到的结果去调用 `start` 钩子函数去创建元素型的 `AST` 节点了。

在源码中， `Vue` 并没有直接去调 `start` 钩子函数去创建 `AST` 节点，而是调用了 `handleStartTag` 函数，在该函数内部才去调的 `start` 钩子函数，为什么要这样做呢？这是因为虽然经过 `parseStartTag` 函数已经把创建 `AST` 节点必要信息提取出来了，但是提取出来的标签属性数组还是需要处理一下，下面我们就来看一下 `handleStartTag` 函数都做了些什么事。 `handleStartTag` 函数源码如下：

```js
function handleStartTag (match) {
    const tagName = match.tagName
    const unarySlash = match.unarySlash

    if (expectHTML) {
      // ...
    }

    const unary = isUnaryTag(tagName) || !!unarySlash

    const l = match.attrs.length
    const attrs = new Array(l)
    for (let i = 0; i < l; i++) {
      const args = match.attrs[i]
      const value = args[3] || args[4] || args[5] || ''
      const shouldDecodeNewlines = tagName === 'a' && args[1] === 'href
        ? options.shouldDecodeNewlinesForHref
        : options.shouldDecodeNewlines
      attrs[i] = {
        name: args[1],
        value: decodeAttr(value, shouldDecodeNewlines)
      }
    }
```

逐行剖析 Vue.js 源码

```
      lastTag = tagName
    }

    if (options.start) {
      options.start(tagName, attrs, unary, match.start, match.end)
    }
  }
```

handleStartTag 函数用来对 parseStartTag 函数的解析结果进行进一步处理，它接收 parseStartTag 函数的返回值作为参数。

handleStartTag 函数的开始定义几个常量：

```js
const tagName = match.tagName        // 开始标签的标签名
const unarySlash = match.unarySlash  // 是否为自闭合标签的标志，自闭合为"",非
const unary = isUnaryTag(tagName) || !!unarySlash  // 布尔值，标志是否为自
const l = match.attrs.length    // match.attrs 数组的长度
const attrs = new Array(l)  // 一个与match.attrs数组长度相等的数组
```

接下来是循环处理提取出来的标签属性数组 match.attrs ，如下：

```js
for (let i = 0; i < l; i++) {
    const args = match.attrs[i]
    const value = args[3] || args[4] || args[5] || ''
    const shouldDecodeNewlines = tagName === 'a' && args[1] === 'href'
    ? options.shouldDecodeNewlinesForHref
    : options.shouldDecodeNewlines
    attrs[i] = {
        name: args[1],
        value: decodeAttr(value, shouldDecodeNewlines)
    }
}
```

上面代码中，首先定义了 args 常量，它是解析出来的标签属性数组中的每一个属性对象，即 match.attrs 数组中每个元素对象。它长这样：

☰ 逐行剖析 Vue.js 源码

接着定义了 value ，用于存储标签属性的属性值，我们可以看到，在代码中尝试取 args 的 args[3] 、 args[4] 、 args[5] ，如果都取不到，则给 value 复制为空

```js
const value = args[3] || args[4] || args[5] || ''
```

接着定义了 shouldDecodeNewlines ，这个常量主要是做一些兼容性处理， 如果 shouldDecodeNewlines 为 true ，意味着 Vue 在编译模板的时候，要对属性值中的换行符或制表符做兼容处理。而 shouldDecodeNewlinesForHref 为 true 意味着 Vue 在编译模板的时候，要对 a 标签的 href 属性值中的换行符或制表符做兼容处理。

```js
const shouldDecodeNewlines = tagName === 'a' && args[1] === 'href'
    ? options.shouldDecodeNewlinesForHref
    : options.shouldDecodeNewlinesconst value = args[3] || args[4] || a
```

最后将处理好的结果存入之前定义好的与 match.attrs 数组长度相等的 attrs 数组中，如下：

```js
attrs[i] = {
    name: args[1],    // 标签属性的属性名，如class
    value: decodeAttr(value, shouldDecodeNewlines) // 标签属性的属性值，如c
}
```

最后，如果该标签是非自闭合标签，则将标签推入栈中（关于栈这个概念后面会说到），如下：

```js
if (!unary) {
    stack.push({ tag: tagName, lowerCasedTag: tagName.toLowerCase(), at
    lastTag = tagName
}
```

如果该标签是自闭合标签，现在就可以调用 start 钩子函数并传入处理好的参数来创建 AST 节点了，如下：

```
}
```

以上就是开始标签的解析以及调用 `start` 钩子函数创建元素型的 `AST` 节点的所有过程。

3.5 解析结束标签

结束标签的解析要比解析开始标签容易多了，因为它不需要解析什么属性，只需要判断剩下的模板字符串是否符合结束标签的特征，如果是，就将结束标签名提取出来，再调用4个钩子函数中的 `end` 函数就好了。

首先判断剩余的模板字符串是否符合结束标签的特征，如下：

```js
const ncname = '[a-zA-Z_][\\w\\-\\.]*'
const qnameCapture = `((?:${ncname}\\:)?${ncname})`
const endTag = new RegExp(`^<\\/${qnameCapture}[^>]*>`)
const endTagMatch = html.match(endTag)

'</div>'.match(endTag)  // ["</div>", "div", index: 0, input: "</div>",
'<div>'.match(endTag)  // null
```

上面代码中，如果模板字符串符合结束标签的特征，则会获得匹配结果数组；如果不合符，则得到null。

接着再调用 `end` 钩子函数，如下：

```js
if (endTagMatch) {
    const curIndex = index
    advance(endTagMatch[0].length)
    parseEndTag(endTagMatch[1], curIndex, index)
    continue
}
```

在上面代码中，没有直接去调用 `end` 函数，而是调用了 `parseEndTag` 函数，关于 `parseEndTag` 函数内部的作用我们后面会介绍到，在这里你暂时可以理解为该函数内部就是去调用了 `end` 钩子函数。

3.6 解析文本

逐行剖析 Vue.js 源码

解析文本也比较容易，在解析模板字符串之前，我们先查找一下第一个 < 出现在什么位置，如果第一个 < 在第一个位置，那么说明模板字符串是以其它5种类型开始的；如果第一个 < 不在第一个位置而在模板字符串中间某个位置，那么说明模板字符串是以文本开头的，那么从开头到第一个 < 出现的位置就都是文本内容了；如果在整个模板字符串里没有找到 < ，那说明整个模板字符串都是文本。这就是解析思路，接下来我们对照源码来了解一下实际的解析过程，源码如下：

```js
let textEnd = html.indexOf('<')
// '<' 在第一个位置，为其余5种类型
if (textEnd === 0) {
    // ...
}
// '<' 不在第一个位置，文本开头
if (textEnd >= 0) {
    // 如果html字符串不是以'<'开头,说明'<'前面的都是纯文本，无需处理
    // 那就把'<'以后的内容拿出来赋给rest
    rest = html.slice(textEnd)
    while (
        !endTag.test(rest) &&
        !startTagOpen.test(rest) &&
        !comment.test(rest) &&
        !conditionalComment.test(rest)
    ) {
        // < in plain text, be forgiving and treat it as text
        /**
           * 用'<'以后的内容rest去匹配endTag、startTagOpen、comment、condit
           * 如果都匹配不上，表示'<'是属于文本本身的内容
           */
        // 在'<'之后查找是否还有'<'
        next = rest.indexOf('<', 1)
        // 如果没有了，表示'<'后面也是文本
        if (next < 0) break
        // 如果还有，表示'<'是文本中的一个字符
        textEnd += next
        // 那就把next之后的内容截出来继续下一轮循环匹配
        rest = html.slice(textEnd)
    }
    // '<'是结束标签的开始 ,说明从开始到'<'都是文本，截取出来
    text = html.substring(0, textEnd)
    advance(textEnd)
}
```

逐行剖析 Vue.js 源码

```
        html = ''
    }
    // 把截取出来的text转化成textAST
    if (options.chars && text) {
        options.chars(text)
    }
```

源码的逻辑很清晰，根据 `<` 在不在第一个位置以及整个模板字符串里没有 `<` 都分别进行了处理。

值得深究的是如果 `<` 不在第一个位置而在模板字符串中间某个位置，那么说明模板字符串是以文本开头的，那么从开头到第一个 `<` 出现的位置就都是文本内容了，接着我们还要从第一个 `<` 的位置继续向后判断，因为还存在这样一种情况，那就是如果文本里面本来就包含一个 `<` ，例如 `1<2</div>` 。为了处理这种情况，我们把从第一个 `<` 的位置直到模板字符串结束都截取出来记作 `rest` ，如下：

```js
let rest = html.slice(textEnd)
```

接着用 `rest` 去匹配以上5种类型的正则，如果都匹配不上，则表明这个 `<` 是属于文本本身的内容，如下：

```js
while (
    !endTag.test(rest) &&
    !startTagOpen.test(rest) &&
    !comment.test(rest) &&
    !conditionalComment.test(rest)
) {

}
```

如果都匹配不上，则表明这个 `<` 是属于文本本身的内容，接着以这个 `<` 的位置继续向后查找，看是否还有 `<` ，如果没有了，则表示后面的都是文本；如果后面还有下一个 `<` ，那表明至少在这个 `<` 到下一个 `<` 中间的内容都是文本，至于下一个 `<` 以后的内容是什么，则还需要重复以上的逻辑继续判断。代码如下：

```js
while (
    !endTag.test(rest) &&
```

```
    .conditionalComment.test(rest)
) {
    // < in plain text, be forgiving and treat it as text
    /**
    * 用'<'以后的内容rest去匹配endTag、startTagOpen、comment、conditionalCor
    * 如果都匹配不上，表示'<'是属于文本本身的内容
    */
    // 在'<'之后查找是否还有'<'
    next = rest.indexOf('<', 1)
    // 如果没有了，表示'<'后面也是文本
    if (next < 0) break
    // 如果还有，表示'<'是文本中的一个字符
    textEnd += next
    // 那就把next之后的内容截出来继续下一轮循环匹配
    rest = html.slice(textEnd)
}
```

最后截取文本内容 `text` 并调用4个钩子函数中的 `chars` 函数创建文本型的 `AST` 节点。

4. 如何保证AST节点层级关系

上一章节我们介绍了 `HTML` 解析器是如何解析各种不同类型的内容并且调用钩子函数创建不同类型的 `AST` 节点。此时你可能会有个疑问，我们上面创建的 `AST` 节点都是单独创建且分散的，而真正的 `DOM` 节点都是有层级关系的，那如何来保证 `AST` 节点的层级关系与真正的 `DOM` 节点相同呢?

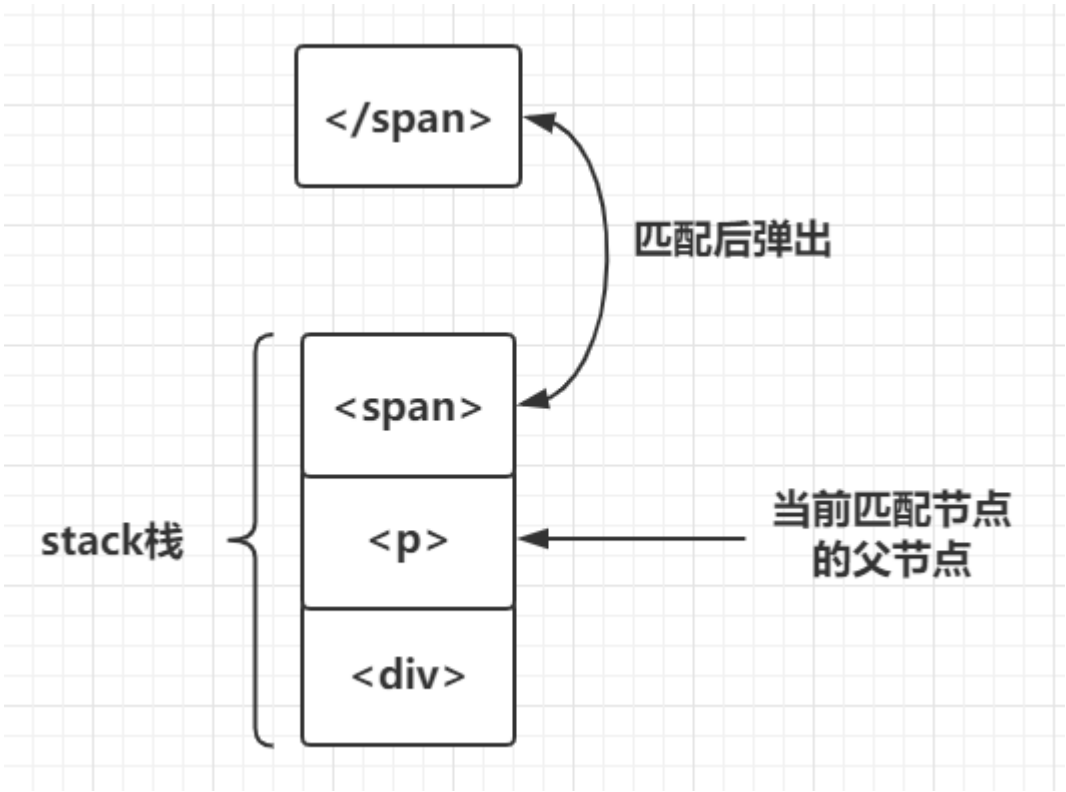
关于这个问题， `Vue` 也注意到了。 `Vue` 在 `HTML` 解析器的开头定义了一个栈 `stack` ，这个栈的作用就是用来维护 `AST` 节点层级的，那么它是怎么维护的呢? 通过前文我们知道， `HTML` 解析器在从前向后解析模板字符串时，每当遇到开始标签时就会调用 `start` 钩子函数，那么在 `start` 钩子函数内部我们可以将解析得到的开始标签推入栈中，而每当遇到结束标签时就会调用 `end` 钩子函数，那么我们也可以在 `end` 钩子函数内部将解析得到的结束标签所对应的开始标签从栈中弹出。请看如下例子:

加入有如下模板字符串:

```html
<div><p><span></span></p></div>
```

当解析到开始标签 `<div>` 时，就把 `div` 推入栈中，然后继续解析，当解析到 `<p>` 时，再把 `p` 推入栈中，同理，再把 `span` 推入栈中，当解析到结束标签 `</span>` 时，此时栈顶的

节点的父节点，如下图：



这样我们就找到了当前被构建节点的父节点。这只是栈的一个用途，它还有另外一个用途，我们再看如下模板字符串：

```html
<div><p><span></p></div>
```

按照上面的流程解析这个模板字符串时，当解析到结束标签 `</p>` 时，此时栈顶的标签应该是 `p` 才对，而现在是 `span` ，那么就说明 `span` 标签没有被正确闭合，此时控制台就会抛出警告：‘tag has no matching end tag.’相信这个警告你一定不会陌生。这就是栈的第二个用途： 检测模板字符串中是否有未正确闭合的标签。

OK，有了这个栈的概念之后，我们再回看上一章 `HTML` 解析器解析不同内容的代码。

5. 回归源码

5.1 HTML解析器源码

以上内容都了解了之后，我们回归源码，逐句分析 `HTML` 解析器 `parseHTML` 函数，函数定义如下：

逐行剖析 Vue.js 源码

```
    var expectHTML = options.expectHTML;
    var isUnaryTag$$1 = options.isUnaryTag || no;
    var canBeLeftOpenTag$$1 = options.canBeLeftOpenTag || no;
    var index = 0;
    var last, lastTag;

    // 开启一个 while 循环，循环结束的条件是 html 为空，即 html 被 parse 完毕
    while (html) {
        last = html;
        // 确保即将 parse 的内容不是在纯文本标签里 (script,style,textarea)
        if (!lastTag || !isPlainTextElement(lastTag)) {
           let textEnd = html.indexOf('<')
              /**
               * 如果html字符串是以'<'开头,则有以下几种可能
               * 开始标签:<div>
               * 结束标签:</div>
               * 注释:<!-- 我是注释 -->
               * 条件注释:<!-- [if !IE] --> <!-- [endif] -->
               * DOCTYPE:<!DOCTYPE html>
               * 需要一一去匹配尝试
               */
            if (textEnd === 0) {
                // 解析是否是注释
                if (comment.test(html)) {

                }
                // 解析是否是条件注释
                if (conditionalComment.test(html)) {

                }
                // 解析是否是DOCTYPE
                const doctypeMatch = html.match(doctype)
                if (doctypeMatch) {

                }
                // 解析是否是结束标签
                const endTagMatch = html.match(endTag)
                if (endTagMatch) {

                }
                // 匹配是否是开始标签
                const startTagMatch = parseStartTag()
                if (startTagMatch) {

```

逐行剖析 Vue.js 源码

```
            let text, rest, next
            if (textEnd >= 0) {

            }
            // 如果在html字符串中没有找到'<'，表示这一段html字符串都是纯文本
            if (textEnd < 0) {
                text = html
                html = ''
            }
            // 把截取出来的text转化成textAST
            if (options.chars && text) {
                options.chars(text)
            }
        } else {
            // 父元素为script、style、textarea时，其内部的内容全部当做纯文本处理
        }

        //将整个字符串作为文本对待
        if (html === last) {
            options.chars && options.chars(html);
            if (!stack.length && options.warn) {
                options.warn(("Mal-formatted tag at end of template: \"'
            }
            break
        }
    }

    // Clean up any remaining tags
    parseEndTag();
    //parse 开始标签
    function parseStartTag() {

    }
    //处理 parseStartTag 的结果
    function handleStartTag(match) {

    }
    //parse 结束标签
    function parseEndTag(tagName, start, end) {

    }
}
```

逐行剖析 Vue.js 源码

- 定义的一些常量和变量
- while 循环
- 解析过程中用到的辅助函数

我们一一来分析：

首先定义了几个常量，如下

```js
const stack = []        // 维护AST节点层级的栈
const expectHTML = options.expectHTML
const isUnaryTag = options.isUnaryTag || no
const canBeLeftOpenTag = options.canBeLeftOpenTag || no    //用来检测一个标
let index = 0    //解析游标，标识当前从何处开始解析模板字符串
let last,   // 存储剩余还未解析的模板字符串
    lastTag  // 存储着位于 stack 栈顶的元素
```

接着开启 `while` 循环，循环的终止条件是 模板字符串 `html` 为空，即模板字符串被全部编译完毕。在每次 `while` 循环中， 先把 `html` 的值赋给变量 `last` ，如下：

```js
last = html
```

这样做的目的是，如果经过上述所有处理逻辑处理过后， `html` 字符串没有任何变化，即表示 `html` 字符串没有匹配上任何一条规则，那么就把 `html` 字符串当作纯文本对待，创建文本类型的 `AST` 节点并且如果抛出异常：模板字符串中标签格式有误。如下：

```js
//将整个字符串作为文本对待
if (html === last) {
    options.chars && options.chars(html);
    if (!stack.length && options.warn) {
        options.warn(("Mal-formatted tag at end of template: \"" + html
    }
    break
}
```

接着，我们继续看 `while` 循环体内的代码：

```
    if (!lastTag || !isPlainTextElement(lastTag)) {

    } else {
      // parse 的内容是在纯文本标签里 (script,style,textarea)
    }
  }
```

在循环体内，首先判断了待解析的 `html` 字符串是否在纯文本标签里，如 `script` , `style` , `textarea` ，因为在这三个标签里的内容肯定不会有 `HTML` 标签，所以我们可直接当作文本处理，判断条件如下：

```js
!lastTag || !isPlainTextElement(lastTag)
```

前面我们说了， `lastTag` 为栈顶元素， `!lastTag` 即表示当前 `html` 字符串没有父节点，而 `isPlainTextElement(lastTag)` 是检测 `lastTag` 是否为是那三个纯文本标签之一，是的话返回 `true` ，不是返回 `fasle` 。

也就是说当前 `html` 字符串要么没有父节点要么父节点不是纯文本标签，则接下来就可以依次解析那6种类型的内容了，关于6种类型内容的处理方式前文已经逐个介绍过，此处不再重复。

5.2 parseEndTag函数源码

接下来我们看一下之前在解析结束标签时遗留的 `parseEndTag` 函数，该函数定义如下：

```js
function parseEndTag (tagName, start, end) {
    let pos, lowerCasedTagName
    if (start == null) start = index
    if (end == null) end = index

    if (tagName) {
      lowerCasedTagName = tagName.toLowerCase()
    }

    // Find the closest opened tag of the same type
    if (tagName) {
      for (pos = stack.length - 1; pos >= 0; pos--) {
        if (stack[pos].lowerCasedTag === lowerCasedTagName) {
          break
```

☰ 逐行剖析 Vue.js 源码

```
      } else {
        // If no tag name is provided, clean shop
        pos = 0
      }

      if (pos >= 0) {
        // Close all the open elements, up the stack
        for (let i = stack.length - 1; i >= pos; i--) {
          if (process.env.NODE_ENV !== 'production' &&
            (i > pos || !tagName) &&
            options.warn
          ) {
            options.warn(
              `tag <${stack[i].tag}> has no matching end tag.`
            )
          }
          if (options.end) {
            options.end(stack[i].tag, start, end)
          }
        }

        // Remove the open elements from the stack
        stack.length = pos
        lastTag = pos && stack[pos - 1].tag
      } else if (lowerCasedTagName === 'br') {
        if (options.start) {
          options.start(tagName, [], true, start, end)
        }
      } else if (lowerCasedTagName === 'p') {
        if (options.start) {
          options.start(tagName, [], false, start, end)
        }
        if (options.end) {
          options.end(tagName, start, end)
        }
      }
    }
  }
```

该函数接收三个参数，分别是结束标签名 `tagName` 、结束标签在 `html` 字符串中的起始和结束位置 `start` 和 `end` 。

这三个参数其实都是可选的，根据传参的不同其功能也不同。

逐行剖析 Vue.js 源码

- 第三种是三个参数都不传递，用于处理栈中剩余未处理的标签

如果 `tagName` 存在，那么就从后往前遍历栈，在栈中寻找与 `tagName` 相同的标签并记录其所在的位置 `pos` ，如果 `tagName` 不存在，则将 `pos` 置为0。如下：

```js
if (tagName) {
    for (pos = stack.length - 1; pos >= 0; pos--) {
        if (stack[pos].lowerCasedTag === lowerCasedTagName) {
            break
        }
    }
} else {
    // If no tag name is provided, clean shop
    pos = 0
}
```

接着当 `pos>=0` 时，开启一个 `for` 循环，从栈顶位置从后向前遍历直到 `pos` 处，如果发现 `stack` 栈中存在索引大于 `pos` 的元素，那么该元素一定是缺少闭合标签的。这是因为在正常情况下， `stack` 栈的栈顶元素应该和当前的结束标签 `tagName` 匹配，也就是说正常的 `pos` 应该是栈顶位置，后面不应该再有元素，如果后面还有元素，那么后面的元素就都缺少闭合标签 那么这个时候如果是在非生产环境会抛出警告，告诉你缺少闭合标签。除此之外，还会调用 `options.end(stack[i].tag, start, end)` 立即将其闭合，这是为了保证解析结果的正确性。

```js
if (pos >= 0) {
    // Close all the open elements, up the stack
    for (var i = stack.length - 1; i >= pos; i--) {
        if (i > pos || !tagName ) {
            options.warn(
                ("tag <" + (stack[i].tag) + "> has no matching end tag.'
            );
        }
        if (options.end) {
            options.end(stack[i].tag, start, end);
        }
    }

    // Remove the open elements from the stack
    stack.length = pos;
    lastTag = pos && stack[pos - 1].tag;
}
```

逐行剖析 Vue.js 源码

最后把 pos 位置以后的元素都从 stack 栈中弹出，以及把 lastTag 更新为栈顶元素：

```js
stack.length = pos;
lastTag = pos && stack[pos - 1].tag;
```

接着，如果 pos 没有大于等于0，即当 tagName 没有在 stack 栈中找到对应的开始标签时， pos 为 -1 。那么此时再判断 tagName 是否为 br 或 p 标签，为什么要单独判断这两个标签呢？这是因为在浏览器中如果我们写了如下 HTML ：

```html
<div>
    </br>
    </p>
</div>
```

浏览器会自动把 </br> 标签解析为正常的
标签，而对于 浏览器则自动将其补全为 ，所以 Vue 为了与浏览器对这两个标签的行为保持一致，故对这两个便签单独判断处理，如下：

```js
if (lowerCasedTagName === 'br') {
    if (options.start) {
        options.start(tagName, [], true, start, end)  // 创建<br>AST节点
    }
}
// 补全p标签并创建AST节点
if (lowerCasedTagName === 'p') {
    if (options.start) {
        options.start(tagName, [], false, start, end)
    }
    if (options.end) {
        options.end(tagName, start, end)
    }
}
```

以上就是对结束标签的解析与处理。

另外，在 while 循环后面还有一行代码：

```js
parseEndTag()
```

`while` 循环，此时就会执行这行代码，这行代码是调用 `parseEndTag` 函数并不传递任何参数，前面我们说过如果 `parseEndTag` 函数不传递任何参数是用于处理栈中剩余未处理的标签。这是因为如果不传递任何函数，此时 `parseEndTag` 函数里的 `pos` 就为0，那么 `pos>=0` 就会恒成立，那么就会逐个警告缺少闭合标签，并调用 `options.end` 将其闭合。

6. 总结

本篇文章主要介绍了 `HTML` 解析器的工作流程以及工作原理，文章比较长，但是逻辑并不复杂。

首先介绍了 `HTML` 解析器的工作流程，一句话概括就是：一边解析不同的内容一边调用对应的钩子函数生成对应的 `AST` 节点，最终完成将整个模板字符串转化成 `AST` 。

接着介绍了 `HTML` 解析器是如何解析用户所写的模板字符串中各种类型的内容的，把各种类型的解析方式都分别进行了介绍。

其次，介绍了在解析器内维护了一个栈，用来保证构建的 `AST` 节点层级与真正 `DOM` 层级一致。

了解了思想之后，最后回归源码，学习了源码中一些处理细节的地方。

在 GitHub 上编辑此页

上次更新: 3/24/2020, 5:37:47 AM

← 模板解析阶段(整体运行流程) 模板解析阶段(文本解析器) →