☰ 逐行剖析 Vue.js 源码

1. 前言

在上篇文章中我们说了，当 `HTML` 解析器解析到文本内容时会调用4个钩子函数中的 `chars` 函数来创建文本型的 `AST` 节点，并且也说了在 `chars` 函数中会根据文本内容是否包含变量再细分为创建含有变量的 `AST` 节点和不包含变量的 `AST` 节点，如下：

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

从上面代码中可以看到，创建含有变量的 `AST` 节点时节点的 `type` 属性为2，并且相较于不包含变量的 `AST` 节点多了两个属性： `expression` 和 `tokens` 。那么如何来判断文本里面是否包含变量以及多的那两个属性是什么呢？这就涉及到文本解析器了，当 `Vue` 用 `HTML` 解析器解析出文本时，再将解析出来的文本内容传给文本解析器，最后由文本解析器解析该段文本里面是否包含变量以及如果包含变量时再解析 `expression` 和 `tokens` 。那么接下来，本篇文章就来分析一下文本解析器都干了些什么。

2. 结果分析

研究文本解析器内部原理之前，我们先来看一下由 `HTML` 解析器解析得到的文本内容经过文本解析器后输出的结果是什么样子的，这样对我们后面分析文本解析器内部原理会有很大的帮助。

及从返回值中取到需要的 `expression` 和 `tokens` 。那么我们就先来看一下 `parseText` 函数如果有返回值，那么它的返回值是什么样子的。

假设现有由 `HTML` 解析器解析得到的文本内容如下：

```js
let text = "我叫{{name}}，我今年{{age}}岁了"
```

经过文本解析器解析后得到：

```js
let res = parseText(text)
res = {
    expression:"我叫"+_s(name)+"，我今年"+_s(age)+"岁了",
    tokens:[
        "我叫",
        {'@binding': name },
        "，我今年"
        {'@binding': age },
        "岁了"
    ]
}
```

从上面的结果中我们可以看到， `expression` 属性就是把文本中的变量和非变量提取出来，然后把变量用 `_s()` 包裹，最后按照文本里的顺序把它们用 `+` 连接起来。而 `tokens` 是个数组，数组内容也是文本中的变量和非变量，不一样的是把变量构造成 `{'@binding': xxx}` 。

那么这样做有什么用呢？这主要是为了给后面代码生成阶段的生成 `render` 函数时用的，这个我们在后面介绍代码生成阶段是会详细说明，此处暂可理解为单纯的在构造形式。

OK，现在我们就可以知道文本解析器内部就干了三件事：

- 判断传入的文本是否包含变量
- 构造expression
- 构造tokens

那么接下来我们就通过阅读源码，逐行分析文本解析器内部工作原理。

3. 源码分析

文本解析器的源码位于 `src/compiler/parser/text-parsre.js` 中，代码如下：

☰ 逐行剖析 Vue.js 源码

```
  const open = delimiters[0].replace(regexEscapeRE, '\\$&')
  const close = delimiters[1].replace(regexEscapeRE, '\\$&')
  return new RegExp(open + '((?:.|\\n)+?)' + close, 'g')
})
export function parseText (text,delimiters) {
  const tagRE = delimiters ? buildRegex(delimiters) : defaultTagRE
  if (!tagRE.test(text)) {
    return
  }
  const tokens = []
  const rawTokens = []
  /**
   * let lastIndex = tagRE.lastIndex = 0
   * 上面这行代码等同于下面这两行代码:
   * tagRE.lastIndex = 0
   * let lastIndex = tagRE.lastIndex
   */
  let lastIndex = tagRE.lastIndex = 0
  let match, index, tokenValue
  while ((match = tagRE.exec(text))) {
    index = match.index
    // push text token
    if (index > lastIndex) {
      // 先把'{{'前面的文本放入tokens中
      rawTokens.push(tokenValue = text.slice(lastIndex, index))
      tokens.push(JSON.stringify(tokenValue))
    }
    // tag token
    // 取出'{{ }}'中间的变量exp
    const exp = parseFilters(match[1].trim())
    // 把变量exp改成_s(exp)形式也放入tokens中
    tokens.push(`_s(${exp})`)
    rawTokens.push({ '@binding': exp })
    // 设置lastIndex 以保证下一轮循环时，只从'}}'后面再开始匹配正则
    lastIndex = index + match[0].length
  }
  // 当剩下的text不再被正则匹配上时，表示所有变量已经处理完毕
  // 此时如果lastIndex < text.length，表示在最后一个变量后面还有文本
  // 最后将后面的文本再加入到tokens中
  if (lastIndex < text.length) {
    rawTokens.push(tokenValue = text.slice(lastIndex))
    tokens.push(JSON.stringify(tokenValue))
  }

```

☰ 逐行剖析 Vue.js 源码

```
    tokens: rawTokens
  }
}
```

我们看到，除开我们自己加的注释，代码其实不复杂，我们逐行分析。

`parseText` 函数接收两个参数，一个是传入的待解析的文本内容 `text` ，一个包裹变量的符号 `delimiters` 。第一个参数好理解，那第二个参数是干什么的呢？别急，我们看函数体内第一行代码：

```js
const tagRE = delimiters ? buildRegex(delimiters) : defaultTagRE
```

函数体内首先定义了变量 `tagRE` ，表示一个正则表达式。这个正则表达式是用来检查文本中是否包含变量的。我们知道，通常我们在模板中写变量时是这样写的：hello 。这里用 `{{}}` 包裹的内容就是变量。所以我们就知道， `tagRE` 是用来检测文本内是否有 `{{}}` 。而 `tagRE` 又是可变的，它是根据是否传入了 `delimiters` 参数从而又不同的值，也就是说如果没有传入 `delimiters` 参数，则是检测文本是否包含 `{{}}` ，如果传入了值，就会检测文本是否包含传入的值。换句话说在开发 `Vue` 项目中，用户可以自定义文本内包含变量所使用的符号，例如你可以使用 `%` 包裹变量如：hello %name%。

接下来用 `tagRE` 去匹配传入的文本内容，判断是否包含变量，若不包含，则直接返回，如下：

```js
if (!tagRE.test(text)) {
    return
}
```

如果包含变量，那就继续往下看：

```js
const tokens = []
const rawTokens = []
let lastIndex = tagRE.lastIndex = 0
let match, index, tokenValue
while ((match = tagRE.exec(text))) {

}
```

☰ 逐行剖析 Vue.js 源码

`null` ，但如果它找到了一个匹配就返回一个数组。例如：

```js
tagRE.exec("hello {{name}}, I am {{age}}")
//返回: ["{{name}}", "name", index: 6, input: "hello {{name}}, I am {{age
tagRE.exec("hello")
//返回: null
```

可以看到，当匹配上时，匹配结果的第一个元素是字符串中第一个完整的带有包裹的变量，第二个元素是第一个被包裹的变量名，第三个元素是第一个变量在字符串中的起始位置。

接着往下看循环体内：

```js
while ((match = tagRE.exec(text))) {
    index = match.index
    if (index > lastIndex) {
      // 先把'{{'前面的文本放入tokens中
      rawTokens.push(tokenValue = text.slice(lastIndex, index))
      tokens.push(JSON.stringify(tokenValue))
    }
    // tag token
    // 取出'{{ }}'中间的变量exp
    const exp = match[1].trim()
    // 把变量exp改成_s(exp)形式也放入tokens中
    tokens.push(`_s(${exp})`)
    rawTokens.push({ '@binding': exp })
    // 设置lastIndex 以保证下一轮循环时，只从'}}'后面再开始匹配正则
    lastIndex = index + match[0].length
  }
```

上面代码中，首先取得字符串中第一个变量在字符串中的起始位置赋给 `index` ，然后比较 `index` 和 `lastIndex` 的大小，此时你可能有疑问了，这个 `lastIndex` 是什么呢？在上面定义变量中，定义了 `let lastIndex = tagRE.lastIndex = 0` ,所以 `lastIndex` 就是 `tagRE.lastIndex` ，而 `tagRE.lastIndex` 又是什么呢？当调用 `exec( )` 的正则表达式对象具有修饰符 `g` 时，它将把当前正则表达式对象的 `lastIndex` 属性设置为紧挨着匹配子串的字符位置，当同一个正则表达式第二次调用 `exec( )` ，它会将从 `lastIndex` 属性所指示的字符串处开始检索，如果 `exec( )` 没有发现任何匹配结果，它会将 `lastIndex` 重置为0。示例如下：

☰ 逐行剖析 Vue.js 源码

```
tagRE.lastIndex    // 14
```

从示例中可以看到， `tagRE.lastIndex` 就是第一个包裹变量最后一个 `}` 所在字符串中的位置。 `lastIndex` 初始值为0。

那么接下里就好理解了，当 `index>lastIndex` 时，表示变量前面有纯文本，那么就把这段纯文本截取出来，存入 `rawTokens` 中，同时再调用 `JSON.stringify` 给这段文本包裹上双引号，存入 `tokens` 中，如下：

```js
if (index > lastIndex) {
    // 先把'{{'前面的文本放入tokens中
    rawTokens.push(tokenValue = text.slice(lastIndex, index))
    tokens.push(JSON.stringify(tokenValue))
}
```

如果 `index` 不大于 `lastIndex` ，那说明 `index` 也为0，即该文本一开始就是变量，例如： `hello` 。那么此时变量前面没有纯文本，那就不用截取，直接取出匹配结果的第一个元素变量名，将其用 `_s()` 包裹存入 `tokens` 中，同时再把变量名构造成 `{'@binding': exp}` 存入 `rawTokens` 中，如下：

```js
// 取出'{{ }}'中间的变量exp
const exp = match[1].trim()
// 把变量exp改成_s(exp)形式也放入tokens中
tokens.push(`_s(${exp})`)
rawTokens.push({ '@binding': exp })
```

接着，更新 `lastIndex` 以保证下一轮循环时，只从 `}}` 后面再开始匹配正则，如下：

```js
lastIndex = index + match[0].length
```

接着，当 `while` 循环完毕时，表明文本中所有变量已经被解析完毕，如果此时 `lastIndex < text.length` ，那就说明最后一个变量的后面还有纯文本，那就将其再存入 `tokens` 和 `rawTokens` 中，如下：

```js
// 当剩下的text不再被正则匹配上时，表示所有变量已经处理完毕
// 此时如果lastIndex < text.length，表示在最后一个变量后面还有文本
```

```
        tokens.push(JSON.stringify(tokenValue))
    }
```

最后，把 `tokens` 数组里的元素用 `+` 连接，和 `rawTokens` 一并返回，如下：

```js
return {
    expression: tokens.join('+'),
    tokens: rawTokens
}
```

以上就是文本解析器 `parseText` 函数的所有逻辑了。

4. 总结

本篇文章介绍了文本解析器的内部工作原理，文本解析器的作用就是将 `HTML` 解析器解析得到的文本内容进行二次解析，解析文本内容中是否包含变量，如果包含变量，则将变量提取出来进行加工，为后续生产 `render` 函数做准备。

在 GitHub 上编辑此页

上次更新: 3/24/2020, 5:37:47 AM

← 模板解析阶段(HTML解析器) 优化阶段 →