☰ 逐行剖析 Vue.js 源码

1. 前言

在前几篇文章中，我们介绍了 Vue 中的虚拟 DOM 以及虚拟 DOM 的 patch (DOM-Diff)过程，而虚拟 DOM 存在的必要条件是得先有 VNode ，那么 VNode 又是从哪儿来的呢？这就是接下来几篇文章要说的模板编译。你可以这么理解：把用户写的模板进行编译，就会产生 VNode 。

2. 什么是模板编译

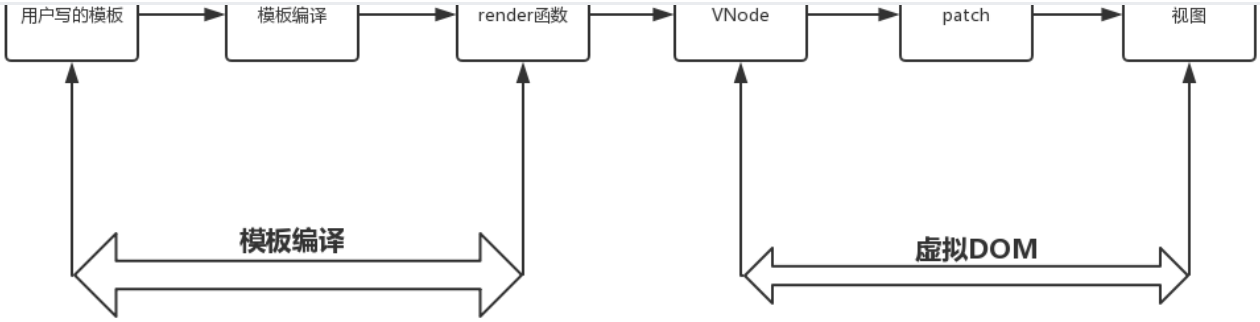
我们知道，在日常开发中，我们把写在 <template></template> 标签中的类似于原生 HTML 的内容称之为模板。这时你可能会问了，为什么说是“类似于原生 HTML 的内容”而不是“就是 HTML 的内容”？因为我们在开发中，在 <template></template> 标签中除了写一些原生 HTML 的标签，我们还会写一些变量插值，如，或者写一些 Vue 指令，如 v-on 、 v-if 等。而这些东西都是在原生 HTML 语法中不存在的，不被接受的。但是事实上我们确实这么写了，也被正确识别了，页面也正常显示了，这又是为什么呢？

这就归功于 Vue 的模板编译了， Vue 会把用户在 <template></template> 标签中写的类似于原生 HTML 的内容进行编译，把原生 HTML 的内容找出来，再把非原生 HTML 找出来，经过一系列的逻辑处理生成渲染函数，也就是 render 函数，而 render 函数会将模板内容生成对应的 VNode ，而 VNode 再经过前几篇文章介绍的 patch 过程从而得到将要渲染的视图中的 VNode ，最后根据 VNode 创建真实的 DOM 节点并插入到视图中， 最终完成视图的渲染更新。

而把用户在 <template></template> 标签中写的类似于原生 HTML 的内容进行编译，把原生 HTML 的内容找出来，再把非原生 HTML 找出来，经过一系列的逻辑处理生成渲染函数，也就是 render 函数的这一段过程称之为模板编译过程。

3. 整体渲染流程

所谓渲染流程，就是把用户写的类似于原生 HTML 的模板经过一系列处理最终反应到视图中称之为整个渲染流程。这个流程在上文中其实已经说到了，下面我们以流程图的形式宏观的了



从图中我们也可以看到，模板编译过程就是把用户写的模板经过一系列处理最终生成 `render` 函数的过程。

4. 模板编译内部流程

那么模板编译内部是怎么把用户写的模板经过处理最终生成 `render` 函数的呢？这内部的过程是怎样的呢？

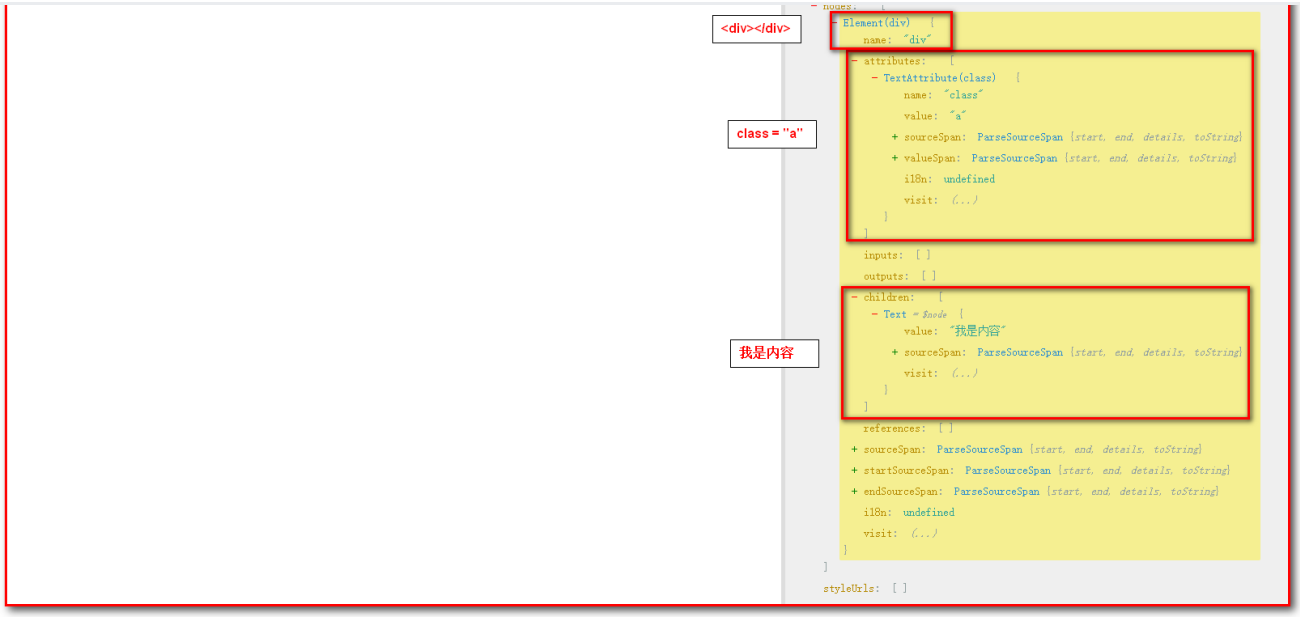
4.1 抽象语法树AST

我们知道，用户在 `<template></template>` 标签中写的模板对 `Vue` 来说就是一堆字符串，那么如何解析这一堆字符串并且从中提取出元素的标签、属性、变量插值等有效信息呢？这就需要借助一个叫做抽象语法树的东西。

所谓抽象语法树，在计算机科学中，**抽象语法树**（AbstractSyntaxTree，AST），或简称**语法树**（Syntax tree），是源代码语法结构的一种抽象表示。它以树状的形式表现编程语言的语法结构，树上的每个节点都表示源代码中的一种结构。之所以说语法是“抽象”的，是因为这里的语法并不会表示出真实语法中出现的每个细节。比如，嵌套括号被隐含在树的结构中，并没有以节点的形式呈现；而类似于if-condition-then这样的条件跳转语句，可以使用带有两个分支的节点来表示。——来自百度百科

我就知道，这段话贴出来也是白贴，因为看了也看不懂，哈哈。那么我们就以最直观的例子来理解什么是抽象语法树。请看下图：

☰ 逐行剖析 Vue.js 源码



从图中我们可以看到，一个简单的 HTML 标签的代码被转换成了一个 JS 对象，而这个对象中的属性代表了这个标签中一些关键有效信息。如图中标识。 有兴趣的同学可以在这个网站在线转换试试：https://astexplorer.net/

4.2 具体流程

将一堆字符串模板解析成抽象语法树 AST 后，我们就可以对其进行各种操作处理了，处理完后用处理后的 AST 来生成 render 函数。其具体流程可大致分为三个阶段：

1. 模板解析阶段：将一堆模板字符串用正则等方式解析成抽象语法树 AST ；
2. 优化阶段：遍历 AST ，找出其中的静态节点，并打上标记；
3. 代码生成阶段：将 AST 转换成渲染函数；

这三个阶段在源码中分别对应三个模块，下面给出三个模块的源代码在源码中的路径：

1. 模板解析阶段——解析器——源码路径： src/compiler/parser/index.js ；
2. 优化阶段——优化器——源码路径： src/compiler/optimizer.js ；
3. 代码生成阶段——代码生成器——源码路径： src/compiler/codegen/index.js ；其对应的源码如下：

```
// 源码位置: /src/complier/index.js

export const createCompiler = createCompilerCreator(function baseCompile
  template: string,
  options: CompilerOptions
): CompiledResult {
```

☰ 逐行剖析 Vue.js 源码

```
    if (options.optimize !== false) {
      // 优化阶段：遍历AST，找出其中的静态节点，并打上标记；
      optimize(ast, options)
    }
    // 代码生成阶段：将AST转换成渲染函数；
    const code = generate(ast, options)
    return {
      ast,
      render: code.render,
      staticRenderFns: code.staticRenderFns
    }
  })

```

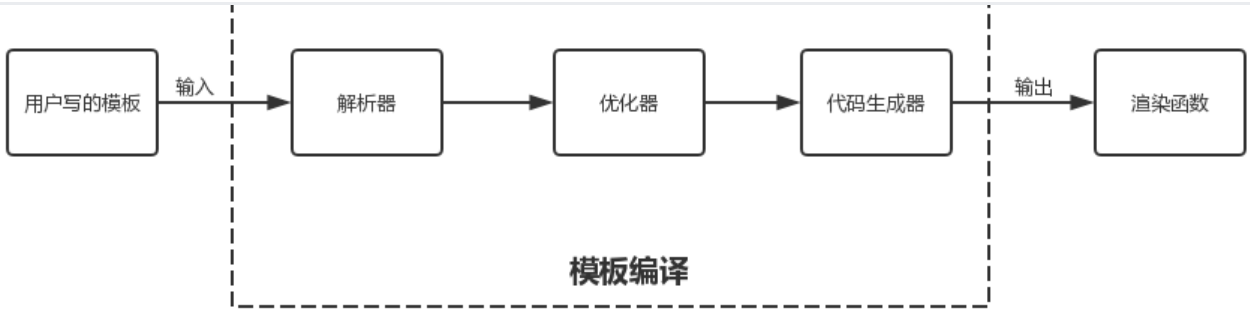
可以看到 `baseCompile` 的代码非常的简短主要核心代码。

- **const ast =parse(template.trim(), options)**: `parse` 会用正则等方式解析 `template` 模板中的指令、 `class` 、 `style` 等数据，形成 `AST` 。
- **optimize(ast, options)**: `optimize` 的主要作用是标记静态节点，这是 `Vue` 在编译过程中的一处优化，挡在进行 `patch` 的过程中， `DOM-Diff` 算法会直接跳过静态节点，从而减少了比较的过程，优化了 `patch` 的性能。
- **const code =generate(ast, options)**: 将 `AST` 转化成 `render` 函数字符串的过程，得到结果是 `render` 函数 的字符串以及 `staticRenderFns` 字符串。

最终 `baseCompile` 的返回值

```
{
    ast: ast,
    render: code.render,
    staticRenderFns: code.staticRenderFns
 }
```

最终返回了抽象语法树(ast)，渲染函数(render)，静态渲染函数(staticRenderFns)，且 `render` 的值为 `code.render` ， `staticRenderFns` 的值为 `code.staticRenderFns` ，也就是说通过 `generate` 处理 `ast` 之后得到的返回值 `code` 是一个对象。



5. 总结

本篇文章首先引出了为什么会有模板编译，因为有了模板编译，才有了虚拟 `DOM` ，才有了后续的视图更新。接着介绍了什么是模板编译，以及介绍了把用户所写的模板经过层层处理直到最终渲染的视图中这个整体的渲染流程；最后介绍了模板编译过程中所需要使用的抽象语法树的概念以及分析了模板编译的具体实施流程，其流程大致分为三个阶段，分别是模板解析阶段、优化阶段和代码生成阶段。那么接下来的几篇文章将会把这三个阶段逐一进行分析介绍。

在 GitHub 上编辑此页

上次更新: 3/24/2020, 5:37:47 AM

← 优化更新子节点

模板解析阶段(整体运行流程) →