Vue编译器

Smount

得到挂载点

获取 options 选项

options 的 render 选项不存在

获取 options 上的 template 选项

template 存在

template 是 string

如果 template 是以 # 开头的说明它是一个 id 选择器

获取它的 innerHTML 内容
template = idToTemplate(template)

如果是开发环境且 template 不存在或为空抛出警告 template 节点是空的或未找到

// idToTemplate(template)
通过 document.querySelector 获取元素的内容

template 的 nodeType 存在

说明该节点是一个普通节点

获取 innerHTML

否则 抛出提示 无效的模版选项

template 不存在 el 节点 存在

获取 outHTML
// innerHTML
template = getOuterHTML(el)

template 存在

编译 template 模版 获取 render 、 staticRenderFns

const { render, staticRenderFns } = compileToFunctions(template, {
// 标记元素在 HTML 模板字符串中的开始和结束索引位置
outputSourceRange: process.env.NODE_ENV !== 'production',
shouldDecodeNewlines,
shouldDecodeNewlinesForHref,
// 界定符号 {{}}
delimiters: options.delimiters,
// 是否保留注释
comments: options.comments
}, this)

将得到的 render 添加到 options 选项上
将得到的 staticRenderFns 添加到 options 选项上

// compileToFunctions
compileToFunctions 是通过 createCompiler(baseOptions) 得到的

const { compile, compileToFunctions } =
createCompiler(baseOptions)

// baseOptions
是一些基本配置对象

export const baseOptions: CompilerOptions = {
expectHTML: true,
// class、style、v-model(input)
modules,
// 指令
directives,
// pre 标签
isPreTag,
// 是否为一元标签
isUnaryTag,
// 一些必须用于 props 的属性
mustUseProp,
// 只有开始标签的标签
canBeLeftOpenTag,
// 保留标签
isReservedTag,
// 命名空间
getTagNamespace,
// 静态 key
staticKeys: genStaticKeys(modules)
}

// createCompiler(baseOptions)
createCompiler 又是通过调用 createCompilerCreator 得到的

export const createCompiler =
createCompilerCreator(function baseCompile (
template:string,
options:CompilerOptions
): CompiledResult {
// 执行 baseOptions 之前的所有事情，只有一个目的，就是构建最终的编译配置
// 核心
// 将 html 模版字符串解析为 ast 对象
const ast = parse(template.trim(), options)
// 优化，比那里 ast，标记静态根节点
if (options.optimize !== false) {
optimize(ast, options)
}
// 代码生成，将 ast 转换成可执行的 render 函数字符串形式
const code = generate(ast, options)
return {
ast,
render: code.render,
staticRenderFns: code.staticRenderFns
}
})

// createCompilerCreator(baseCompile:Function)
参数是一个函数 返回也是一个函数，这里返回的函数其实就是 createCompiler

// createCompiler (baseOptions) 这里的 baseOptions 参考备注

定义一个 compile 函数

return {
compile,
compileToFunctions:createCompileToFunctionFn(compile)
}

// compile (
// 字符串模版
template: string,
// 编译选项
options?: CompilerOptions
)

// 基于平台现有的编译选项为原型创建最终的编译配置
const finalOptions = Object.create(baseOptions)

options 存在

重写 warn 函数

如果 options.modules 存在，将 baseOptions.modules 和 options.modules 合并

如果 options.directives 存在，将 baseOptions.directives 和 options.modules 合并

遍历 options 如果 key 不等于 modules 和 directives 将选项赋值给 finalOptions

定义 errors 数组
定义 tips 数组
定义一个 warn 的增强方法

判断 options 是否存在

将增强的 warn 方法 添加到 finalOptions 上

执行 baseCompile 函数得到编译结果
const compiled = baseCompile(template.trim(),
finalOptions)

将执行期间产生的 errors 和 tips 添加到 compiled 对象上

// createCompileToFunctionFn(compile)

const cache = Object.create(null)

返回函数 compileToFunctions(
// 字符串模版
template: string,
// 编译选项
options?: CompilerOptions,
// 组件实例
vm?: Component
)

获取 options，warn

删除 options 上的 warn 方法

const key = options.delimiters
? String(options.delimiters) + template
: template

判断 cache 有无当前对应的编译好的数据如果有直接返回缓存内的

执行 compile 函数，得到编译结果
const compiled = compile(template, options)

// 通过 new Function(code) 将字符串转换成函数
res.render = createFunction(compiled.render, fnGenErrors)

返回编译结果并将结果缓存
return (cache[key] = res)

// parse(template,options)

获取 options 上的相关属性

获取保留标签

获取 style class model 模块中导出的函数
transforms = pluckModuleFunction(options.modules, 'transformNode')

preTransforms = pluckModuleFunction(options.modules, 'preTransformNode')

postTransforms = pluckModuleFunction(options.modules, 'postTransformNode')

定义 存放 标签 ast 对象的 数组

定义是否保留空白标签

定义 空白元素的选项

定义最后 return 出去的 root ast 对象

记录当前的父元素及相关配置信息

定义了 closeElement 函数

定义了 trimEndingWhitespace 函数

定义了 checkRootConstraints 函数

//baseCompile
// 执行 baseOptions 之前的所有事情，只有一个目的，就是构建最终的编译配置

// 将 html 模版字符串解析为 ast 对象

如果 options.optimize 为 false 则进行优化处理，比当前的 ast 对象，标记静态根节点

// 代码生成，将 ast 转换成可执行的 render 函数字符串形式

/**
* 主要做了 3 件事：
* 1、如果元素没有被处理过，即 el.processed 为 false，则调用 processElement 方法处理节点上的众多属性
* 2、让自己和父元素产生关系，将自己放到父元素的 children 数组中，并设置自己的 parent 属性为 currentParent
* 3、设置自己的子元素，将自己所有非插槽的子元素放到自己的 children 数组中
*/
closeElement(element)

清理节点末尾的空白字符

如果标签不在 pre 标签内且未被标记为已处理 执行 processElement(element, options) 处理标签

存放 ast 对象数组有值，且 当前标签的 ast 对象不是 root
如果 根节点有 v-if 指令必须要同级别要提供一个含有 v-else-if 或者 v-else 的节点 processIfConditions(element, currentParent)

如果父节点存在且当前节点可访问
如果当前节点存在 else-if 或者 else 指令 processIfConditions(element, currentParent)
否则 如果当前节点为作用域插槽
获取当前节点的作用域名称
将元素放到父元素的肚子里 currentParent.children.push(element)
在自己身上记录 parent 属性，标记自己的父元素是谁

/**
* 删除元素中空白的文本节点，比如： ，删除 div 元素中的空白节点，将其从元素的 children 属性中移出去
*/
trimEndingWhitespace(el)

如果不在 pre 标签内

定义 lastNode

用 while 循环为 lastNode 赋值 当前标签节点的 children 最后一个静态节点且内容为 " 然后弹出 children 最后一个节点

/**
* 检查根元素：
* 不能使用 slot 和 template 标签作为组件的根元素
* 不能在有状态组件的 根元素 上使用 v-for 指令，因为它会渲染出多个元素
* @param {*} el
*/
checkRootConstraints(el)

如果当前节点的标签名称为 slot 或者 template 抛出警告

如果当前 attrsMap 有 v-for 抛出警告

// parse(template,options)

执行 parseHTML 传入HTML模版和相关选项这里还定义了 start、end、chars、comment 方法，在parseHTML 方法中会用到

返回 root

定义存放标签属性的对象数组

判断 html 是否存在的 while 循环

last = html

判断标签在不在 script style textarea 标签内

如果 html 等于 last 则处理完成
如果 options.chars 属性存在并且处理 options.chars(html)

清理剩余标签
parseEndTag()

//处理 script style textarea 的内容
如果在 script style textarea 标签内，匹配开始位置和结束位置，得到位置将内容置为空，再将这部分内容剪切掉

parseEndTag(stackedTag, index - endTagLength, index)

// 保证处理的内容不在 script style textarea 的标签内
如果不在 script style textarea 标签内

匹配 '<' 标签

如果 '<' 是标签开始位置，则为注释
如果符合注释的匹配规则
获取注释的结尾位置
如果注释结尾的索引大于等于 0 且 options.shouldKeepComment 为 true 即需要保留注释
给 options 的 comment 属性将注释内容添加进去
剪切处理过的内容
advance(commentEnd + 3)

/**
* 通过循环遍历 html 模版字符串，依次处理其中的各个标签，以及标签上的属性
* @param {*} html html 模版
* @param {*} options 配置项
*/
parseHTML(html,options)

定义 函数 advanve、parseStartTag、handleStartTag、parseEndTag

/**
* 重置 html，html = 从索引 n 位置开始的向后的所有字符
* index 为 html 在 原始的 模版字符串 中的的开始索引，也是下一次该处理的字符开始位置
* @param {*} n 索引
*/
advanve(n)

/**
* 解析开始标签，比如：
* @returns { tagName: 'div', attrs: [[xx], ...], start: index }
*/
parseStartTag()

//进一步处理开始标签 例如：<Start><Start/> 得到的属性
handleStartTag (match)

/**
* 解析结束标签，比如：
* 最主要的事就是：
* 1、处理 stack 数组，从 stack 数组中找到当前结束标签对应的开始标签，然后调用 options.end 方法
* 2、处理完结束标签之后调整 stack 数组，保证在正常情况下 stack 数组中的最后一个元素就是下一个结束标签对应的开始标签
* 3、处理一些异常情况，比如 stack 数组最后一个元素不是当前结束标签对应的开始标签，还有就是
* br 和 p 标签单独处理
* @param {*} tagName 标签名，比如 div
* @param {*} start 结束标签的开始索引
* @param {*} end 结束标签的结束索引
*/
parseEndTag (tagName, start, end)

// 处理结束节点
end(tag,start,end)

获取存放 ast 对象的 stack ， 拿到最后一个元素

将获取的元素的元素从存放 ast 对象的数组中弹出

currentParent = stack[stack.length - 1]

closeElement(element)

//处理文本节点
chars(text,start,end)

如果父节点不存在，直接丢弃文本 结束函数执行

处理 ie 浏览器标签为 textarea 且 文本内容等于 placeholder 直接结束函数执行

将内容放到父节点的 children 中

如果是 pre 标签，获取文本

如果 children 不存在 将 text 赋值为 ''

如果空白选项存在
且空白选项值为 condense 则进行正则匹配符合的值将其赋值给 text
否则 text = ' '
如果空白选项不存在 text = preserveWhitespace ? ' ' : ''

如果 text 存在

判断是否为 pre 标签且空白选项不为 codense，根据指定正则将匹配文本替换为 ' '

定义 res ， child

如果 标签不是 pre 标签且文本不为 ''
child = {
type: 2,
expression: res.expression,
tokens: res.tokens,
text
}

如果文本为空或者父元素没有子节点或者父节点的最后一个子节点的文本内容为空
child = {
type: 3,
text
}

child 存在，则将 child 放到父元素的肚子里，即 currentParent.children 数组中

// 处理注释内容
comment(text,start,end)

如果 currentParent 存在

const child: ASTText = {
type: 3,
text,
isComment: true
}

currentParent.children.push(child)

/**
* @param {*} tag 标签名
* @param {*} attrs 属性 [{ name: attrName, value: attrValue, start, end }, ...]
* @param {*} unary 是否为自闭合标签
* @param {*} start 标签开始索引位置
* @param {*} end 标签结束索引位置
*/
start (tag, attrs, unary, start, end)

如果 ns 存在 则将 ns 添加到当前标签的 ast 对象 element 上

遍历 preTransforms 执行 preTransforms[i](element, options)，即处理带有 v-model 指令的 input 标签，以及标签上的众多属性，比如 v-for、v-if、type 其他指令属性等等，最后将结果都记录在 element 对象上

处理 v-pre 指令

判断是否为平台的 pre 标签 如果是 inPre = true

如果 inPre 为 true 执行 processRawAttrs(element)

如果 当前标签的 ast 对象 未被标记为已处理，则再处理一次标签

else if (!element.processed) {
// structural directives
processFor(element)
processIf(element)
processOnce(element)
}

如果 root 不存在 将当前标签的 ast 对象 赋值给 root

如果标签为非被和标签，将当前标签的 ast 对象赋值给 currentParent

如果是自闭合标签执行 closeElement(element)

获取命名空间

const ns = (currentParent && currentParent.ns) || platformGetTagNamespace(tag)

处理 ie 浏览器的情况

生成当前标签的 ast 对象

let element: ASTElement = createASTElement(tag, attrs, currentParent)

如果 ns 存在 则将 ns 添加到当前标签的 ast 对象 element 上

遍历 preTransforms 执行 preTransforms[i](element, options)，即处理带有 v-model 指令的 input 标签，以及标签上的众多属性，比如 v-for、v-if、type 其他指令属性等等，最后将结果都记录在 element 对象上

处理 v-pre 指令
processPre

获取当前 ast 对象 v-pre 属性，并将 attrsMap 上的 v-pre 属性删除

若获取的 v-pre 属性值 存在 则 el.pre = true

/是否为平台的 pre 的标签
platformIsPreTag

在 parse 函数过程中就已经为此赋值

// 处理标签自带的属性
/**
* 设置 el.attrs 数组对象，每个元素都是一个属性对象 { name: attrName, value: attrVal, start, end }
*/
processRawAttrs

// processRawAttrs

获取 ast 对象上的 attrsList
const list = el.attrsList

获取 attsList 长度
const len = list.length

如果长度存在
基于长度为 len 建立一个数组
遍历 list 生成一个新数组成员为
{ name: attrName, value: attrVal, start, end }

如果不存在
将节点上的 pre 属性赋值为 true

// 处理 v-for 指令
/**
* 处理 v-for，将结果设置到 el 对象上，得到：
* el.for = 可迭代对象，比如 arr
* el.alias = 别名，比如 item
* @param {*} el 元素的 ast 对象
*/
processFor

// processFor

定义表达式 exp

获取节点上的 exp 且 exp 存在

const res = parseFor(exp)

如果 res 存在,基于 el ast 对象和 res 合并生成一个新的对象

如果不存在抛出警告 v-for 需要迭代对象

/**
* 解析 v-for指令的表达式子，得到 { for : iterator （数组），alias : 别名 （item） }
* @export
* @param {string} exp for 表达式
*/
parseFor(exp)

// parseFor

通过制定的正则去匹配得到 迭代对象

const inMatch = exp.match(forAliasRE)

如果迭代对象不存在 结束函数执行

如果存在 res.for = exp

通过指定的正则去匹配别名

const alias = inMatch[1].trim().replace(stripParensRE, '')

通过指定的正则去匹配迭代器

const iteratorMatch = alias.match(forIteratorRE)

如果迭代器还是迭代器对象继续匹配

res.alias = alias.replace(forIteratorRE, '').trim()
res.iterator1 = iteratorMatch[1].trim()
if (iteratorMatch[2]) {
res.iterator2 = iteratorMatch[2].trim()
}

返回 res

// 处理 v-if 指令
processIf

// processIf

获取 v-if 属性的值，并在 attrsMap 中删除 v-if
const exp = getAndRemoveAttr(el, 'v-if')

如果 exp 存在,将当前节点的 if 属性赋值为 exp ， 并将 exp 添加节点上的 ifConditions 数组中，若 ifConditions 不存在将其赋值为空数组
el.if = exp
addIfCondition(el, {
exp: exp,
block: el
})

/**
* 将传递进来的条件对象放进 el.ifConditions 数组中
*/
addIfCondition

判断当前 ast 对象上 ifConditions 是否存在
若存在直接将 对象添加到 ifConditions 中
否则将其赋值为空数组

// 处理 v-once 指令
processOnce

// processOnce

获取 v-once 属性，并将 attrsMap 中的 v-once 属性删除
const once = getAndRemoveAttr(el, 'v-once')

若 once 存在
el.once = true

//编译器优化
// optimize

// 生成渲染函数
// genlerate

//返回当前实例
return mount.call(this, el, hydrating)