第一步是将 模板字符串 转换成 element ASTs（解析器）

第二步是对 AST 进行静态节点标记，主要用来做虚拟DOM的渲染优化（优化器）

第三步是 使用 element ASTs 生成 render 函数代码字符串（代码生成器）

<div><p>{{name}}</p></div>

模板转成AST后

```
var o = {
    tag: "div"
    type: 1,
    staticRoot: false,
    static: false,
    plain: true,
    parent: undefined,
    attrsList: [],
    attrsMap: {},
    children: [{
        tag: "p"
        type: 1,
        staticRoot: false,
        static: false,
        plain: true,
        parent: {
            tag: "div",
            ...
        },
        attrsList: [],
        attrsMap: {},
        children: [{
            type: 2,
            text: "{{name}}",
            static: false,
            expression: "_s(name)"
        }]
    }]
}
```

先分析转换成AST的基本原理:

1.基于< 符号来分割, 匹配到后, 记录匹配的长度, 然后利用 advance函数一小段一小段的截取

2. 先排除各种注释, 再用各种正则match标签名, 属性, 匹配到后把结果放在 parseTag 或者 start 函数里面进一步处理

3.维护一个stack (栈的数据结果, 后进先出), 这个 stack 是用来记录一个层级关系的，用来记录DOM的深度。

更准确的说，当解析到一个 开始标签 或者 文本，无论是什么， stack 中的最后一项，永远是当前正在被解析的节点的 parentNode 父节点。

通过 stack 解析器就可以把当前解析到的节点 push 到 父节点的 children 中。

也可以把当前正在解析的节点的 parent 属性设置为 父节点。

事实上也确实是这么做的。

具体流程

以 < 开头的字符串有以下几种情况：

开始标签 <div>

结束标签 </div>

HTML注释 <!-- 我是注释 -->

Doctype <!DOCTYPE html>

条件注释（Downlevel-revealed conditional comment）

当然我们解析器在解析的过程中遇到的最多的是 开始标签 结束标签 和 注释

截取文本

核心代码

```
var last, lastTag;
while (html) {
  last = html;
  // Make sure we're not in a plaintext content element like script/style
  if (!lastTag || !isPlainTextElement(lastTag)) {
    var textEnd = html.indexOf('<');
    if (textEnd === 0) {
      // Comment:
      if (comment.test(html)) {…

      // http://en.wikipedia.org/wiki/Conditional_comment#Downlevel-revealed_conditional_comment
      if (conditionalComment.test(html)) {…

      // Doctype:
      var doctypeMatch = html.match(doctype);
      if (doctypeMatch) {…

      // End tag:
      var endTagMatch = html.match(endTag);
      if (endTagMatch) {
        var curIndex = index;
        advance(endTagMatch[0].length);
        parseEndTag(endTagMatch[1], curIndex, index);
        continue
      }

      // Start tag:
      var startTagMatch = parseStartTag();
      if (startTagMatch) {
        handleStartTag(startTagMatch);
        if (shouldIgnoreFirstNewline(lastTag, html)) {
          advance(1);
        }
        continue
      }
    }
```

textEnd = html.inexOf('<');

if(textEnd == 0)

调用了一个工具函数 advance

function advance (n) {
index += n;
html = html.substring(n);
}

这里index是全局变量
html是剩余的模板字符串,
每次从剩余的字符串中截掉 n 个字符
因为这n个字符已经被处理了

如果是注释, 处理注释
if (comment.test(html)) {
var commentEnd = html.indexOf('-->');

if (commentEnd >= 0) {

advance(commentEnd + 3);
continue

}

调用的正则

const doctype = /^<!DOCTYPE [^>]+>/i
const comment = /^<!--/
const conditionalComment = /^<!\[/

比较简单, 注意一个简单的技巧
[^>]+ 只要后面字符不是右尖括号, 就加入到匹配结果中. 这样就能获取<......> 尖括号所有的内容

调用了parseTagEnd函数

1. 如果endTag标签在 stack里面, 执行出栈操作, 因为该标签的作用域其实已经结束, 后面的标签不会是它的子节点

如果是结束标签, 处理结束标签

var endTagMatch = html.match(endTag);

if (endTagMatch) {
var curIndex = index;
advance(endTagMatch[0].length);
parseEndTag(endTagMatch[1], curIndex, index);

continue
}

调用的正则

const ncname = '[a-zA-Z_][\\w\\-\\.]*'
const qnameCapture = '((?:' + ncname + '\\:)?' + ncname + ')'
endTag = new RegExp('^<\\/' + qnameCapture + '[^>]*>')
如：
</svg:path> </div> </div >都可以匹配到

</div>匹配的结果是 一个数组 ["</div>", "div", index: 0, input: "</div>"]
数组第一个是匹配的全部, 第二是正则的第一个子项, 也就小括号的内容(标签名 div)

?:的意思是这个小括号的内容需要匹配但是不用捕获, 提高正则性能

如果是开始标签, 处理开始标签
处理开始标签比较复杂, 还需要处理各种属性，自定义属性等等

var startTagMatch = parseStartTag();

if (startTagMatch) {
handleStartTag(startTagMatch);

if (shouldIgnoreFirstNewline(lastTag, html)) {
advance(1);
}
continue
}

首先获取到标签名, tagName:div
然后截掉 标签 <div 开始这部分

while循环, 只要还没到标签结束, 就不断获取div中的属性, 存储到attrs属性中, 在跳过已处理的属性

具体如何处理标签
参考标签处理部分

```
function parseStartTag () {
  var start = html.match(startTagOpen);
  if (start) {
    var match = {
      tagName: start[1],
      attrs: [],
      start: index
    };
    advance(start[0].length);
    var end, attr;
    while (!(end = html.match(startTagClose))
        && (attr = html.match(attribute))) {

      advance(attr[0].length);
      match.attrs.push(attr);
    }
    if (end) {
      match.unarySlash = end[1];
      advance(end[0].length);
      match.end = index;
      return match
    }
  }
}
```

调用的正则

const ncname = '[a-zA-Z_][\\w\\-\\.]*'
const qnameCapture = '((?:' + ncname + '\\:)?' + ncname + ')'

startTagOpen = new RegExp('^<' + qnameCapture)

startTagClose = /^\s*(\/?)>/, 包含: > 或 />

attribute = /^\s*([^\s"'<>\/=]+)(?:\s*(=)\s*(?:"([^"]*)"+|'([^']*)'+|([^\s"'=<>`]+)))?/;

attribute看起来有点吓人, 其实分解开来看就容易一些

比如 title='logo' 或者 title="logo" 或者 title = logo

([^\s"'<>\/=]+) 匹配的是 title

(?:\s*(=) 匹配的是 =，前后可以有空格

(?:"([^"]*)"+|'([^']*)'+|([^\s"'=<>`]+)) 匹配的是 "logo" 或者 'logo' 或者logo

上面几种情况如果匹配到, continue跳出本次循环, 如果没有匹配到任何结果, 继续往下执行

进入文本解析

来到这里, 说明不是以标签 或者 注释开头, 那么有可能是这几种情况:

1. <文本<span>...
2. abc<bdf<span>
3.abc<span>

while(当不是标签 和 注释),
以<为参考, 一小段一小段截取,
截取<左边的文本,
剩余的rest在去判断一下是不是标签注释

当循环完之后

textEnd 累加完成,
说明剩余的html从0到textEnd都是文本

rest (<span>..)则是以标签或者注释开头的剩余字符串

提取text (abc<bdf), 并且跳过已处理的字段

如果textEnd<0, 说明已经剩余的html没有左尖括号了
则剩余的都归为文本

如果配置传入了处理文本函数 chars, 则调用 chars

如何处理文本, 参考"标签处理部分"

```
var text = (void 0), rest = (void 0), next = (void 0);
if (textEnd >= 0) {
  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
    next = rest.indexOf('<', 1);
    if (next < 0) { break }
    textEnd += next;
    rest = html.slice(textEnd);
  }
  text = html.substring(0, textEnd);
  advance(textEnd);
}

if (textEnd < 0) {
  text = html;
  html = '';
}

if (options.chars && text) {
  options.chars(text);
}
```

void 0, 实际上就是 undefined 意思
为了避免und..被重写

模板解析为ast的对比图

```
<div id="app">
</div>
<script>
    var e = new Vue({
        el: '#app',

        template: '<div id="app">\
            这里是文本<箭头之后的文本<添加一点文字\
            <a :href="url" target="_blank" >前面的文本{{title}}后面的文本</a>\
            <img :src="img" />\
            <span @click="add">add</span>\
            <span @click="changeA">changeA</span>\
            <div v-if="show">show</div>\
            <ul v-for = "k in arr"><li>{{k}}</li></ul>\
            arr: {{arr}} <br/>\
            a:  {{a}} <br/>\
            result: {{result}}\
          </div>',
        data: {
            url: 'https://www.imliutao.com',
            title: '刘涛的个人小站',
            img: 'https://pic1.zhimg.com/092406f3919e915fffc7ef2f2410e560_is.jpg',
            arr: [1, 2],
            show: false,
            a: 3,
            obj: {
                a: 1,
                b: 2
            }
        }
```

```
ast
▼{type: 1, tag: "div", attrsList: Array(1), attrsMap: {…}, parent: undefined, …}
  ▼attrs: Array(1)
    ▶0: {name: "id", value: ""app""}
     length: 1
    ▶__proto__: Array(0)
  ▶attrsList: [{…}]
  ▶attrsMap: {id: "app"}
  ▼children: Array(17)
    ▶0: {type: 3, text: "                这里是文本<箭头之后的文本<添加一点文字
    ▶1: {type: 1, tag: "a", attrsList: Array(2), attrsMap: {…}, parent: {…}, …}
    ▶2: {type: 3, text: " "}
    ▼3:
      ▼attrs: Array(1)
        ▶0: {name: "src", value: "img"}
         length: 1
        ▶__proto__: Array(0)
      ▶attrsList: [{…}]
      ▶attrsMap: {:src: "img"}
      ▶children: []
       hasBindings: true
      ▶parent: {type: 1, tag: "div", attrsList: Array(1), attrsMap: {…}, parent: undefined,
       plain: false
       tag: "img"
       type: 1
      ▶__proto__: Object
    ▶4: {type: 3, text: " "}
    ▶5: {type: 1, tag: "span", attrsList: Array(1), attrsMap: {…}, parent: {…}, …}
    ▶6: {type: 3, text: " "}
    ▶7: {type: 1, tag: "span", attrsList: Array(1), attrsMap: {…}, parent: {…}, …}
    ▶8: {type: 3, text: " "}
    ▶9: {type: 1, tag: "div", attrsList: Array(0), attrsMap: {…}, parent: {…}, …}
```

ast只是把模板转化成一个树形结构的对象, 并没有把数据解析到模板中; 几个关键属性

hasBindings 是否有绑定数据, 那img是绑定有数据的
expression: "" a: "+_s(a)+" "", 是解析{{a}} 之后的结果, _s(a) 用来生成创建文本节点

生成ast之后, 进行静态资源优化, 这是为了vnode中diff优化准备的.

优化器的目标：遍历生成的模板AST树

检测纯静态的子树，即部分永远不需要改变的DOM。

一旦我们检测到这些子树，我们可以：

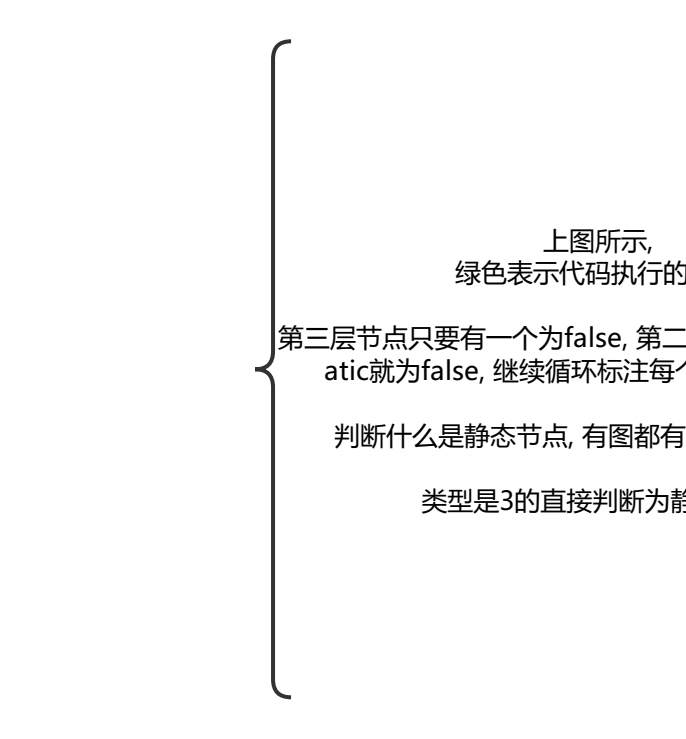
1。把它们变成常数，这样我们就不再需要在每个重新渲染上为它们创建新节点；

2。在patch的diff中跳过这些节点。

原理是使用了两个递归, 从根节点开始, 循环下面的每个子节点 n1, n2 , n3 ...;
如果n1又存在自己的子节点 n11, n22, n33, n44

则先看这几个节点是否是静态节点, 如果都是, 就回归标注n1为 静态节点
如果有一个不是, 就回归标注n1为 非静态节点

标注静态根节点也是同样的流程, 分别调用了以下两个方法



```
function markStatic$1 (node) {
  node.static = isStatic(node);
  if (node.type === 1) {
    // do not make component slot content static. this avoids
    // 1. components not able to mutate slot nodes
    // 2. static slot content fails for hot-reloading
    if (
      !isPlatformReservedTag(node.tag) &&
      node.tag !== 'slot' &&
      node.attrsMap['inline-template'] == null
    ) {
      return
    }
    for (var i = 0, l = node.children.length; i < l; i++) {
      var child = node.children[i];
      markStatic$1(child);
      if (!child.static) {
        node.static = false;
      }
    }
    if (node.ifConditions) {
      for (var i$1 = 1, l$1 = node.ifConditions.length; i$1 < l$1; i$1++) {
        var block = node.ifConditions[i$1].block;
        markStatic$1(block);
        if (!block.static) {
          node.static = false;
        }
      }
    }
  }
}
```

上图所示,
绿色表示代码执行的顺序

第三层节点只要有一个为false, 第二层的第一个节点的static就为false, 继续循环标注每个子节点的static

判断什么是静态节点, 有图都有注释, 一目了然;

类型是3的直接判断为静态节点

```
function isStatic (node) {
  if (node.type === 2) { // expression
    return false
  }
  if (node.type === 3) { // text
    return true
  }
  return !!(node.pre || (
    !node.hasBindings && // no dynamic bindings
    !node.if && !node.for && // not v-if or v-for or v-else
    !isBuiltInTag(node.tag) && // not a built-in
    isPlatformReservedTag(node.tag) && // not a component
    !isDirectChildOfTemplateFor(node) &&
    Object.keys(node).every(isStaticKey)
  ))
}
```

这里也是递归, 只是父节点没有用到处理子节点后回归的值, 直接取判断是否是静态根节点

判断静态根节点的条件是：

当前节点是静态节点，并且有子节点，并且子节点不是单个 静态文本节点这种情况

把一个只包含静态文本的节点标记为根节点，那么它的成本会超过收益~

也就是说如果只是一个静态文本节点, 没必要转成vnode, 直接每次更新文本就好, 当做常量处理

```
function markStaticRoots (node, isInFor) {
  if (node.type === 1) {
    if (node.static || node.once) {
      node.staticInFor = isInFor;
    }
    // For a node to qualify as a static root, it should have children that
    // are not just static text. Otherwise the cost of hoisting out will
    // outweigh the benefits and it's better off to just always render it fresh.
    if (node.static && node.children.length && !(
      node.children.length === 1 &&
      node.children[0].type === 3
    )) {
      node.staticRoot = true;
      return
    } else {
      node.staticRoot = false;
    }
    if (node.children) {
      for (var i = 0, l = node.children.length; i < l; i++) {
        markStaticRoots(node.children[i], isInFor || !!node.for);
      }
    }
    if (node.ifConditions) {
      for (var i$1 = 1, l$1 = node.ifConditions.length; i$1 < l$1; i$1++) {
        markStaticRoots(node.ifConditions[i$1].block, isInFor);
      }
    }
  }
}
```

静态优化之后, 每个节点多了两个属性

static 和 staticRoot

然后继续处理, 把ast 变成可执行的render函数, 类型下面

{render: "with(this){return _c('div',{attrs:{"id":"app"}},[_]}