



## 复习

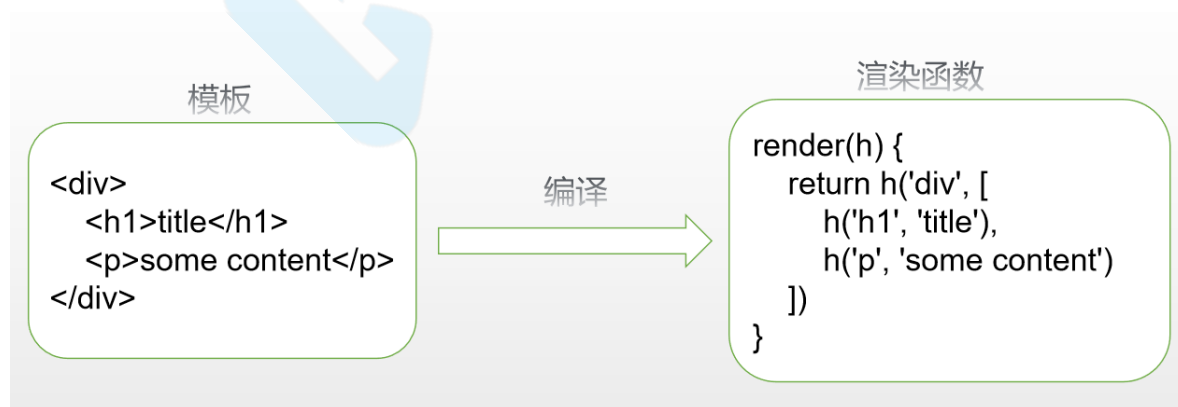
- 虚拟DOM: 利用patching算法转换虚拟DOM为DOM

测试用例: examples\test\04-vdom.html

- diff
  - w 是什么
  - w 性能、跨平台、兼容性
  - w 在什么地方, patch, 存在新旧虚拟dom
  - h 怎么执行?
    - 深度优先, 同级比较
    -

## 模板编译

模板编译的主要目标是**将模板(template)转换为渲染函数(render)**



template => render()

## 模板编译必要性

Vue 2.0需要用到VNode描述视图以及各种交互, 手写显然不切实际, 因此用户只需编写类似HTML代码的Vue模板, 通过编译器将模板转换为可返回VNode的render函数。

## 体验模板编译

开课吧web全栈架构师

带编译器的版本中，可以使用template或el的方式声明模板，06-1-compiler.html

```
(function anonymous(
) {
  with(this){return _c('div',{attrs:{"id":"demo"}},[_c('h1',[_v("Vue模板编译")]),_v(" ")],_c('p',[_v(_s(foo))]),_v(" ")],_c('comp')],1)}
})
```

输出结果大致如下：

```
(function anonymous() {
  with(this){return _c('div',{attrs:{"id":"demo"}},[
    _c('h1',[_v("Vue模板编译")]),
    _v(" "),_c('p',[_v(_s(foo))]),
    _v(" "),_c('comp')],1)}
})
```

元素节点使用createElement创建，别名\_c

本文节点使用createTextVNode创建，别名\_v

表达式先使用toString格式化，别名\_s

其他渲染helpers: src\core\instance\render-helpers\index.js

## 整体流程

### compileToFunctions

若指定template或el选项，则会执行编译，platforms\web\entry-runtime-with-compiler.js

### 编译过程

编译分为三步：解析、优化和生成，src\compiler\index.js

## 解析模板编译过程

实现模板编译共有三个阶段：解析、优化和生成

### 解析 - parse

解析器将模板解析为抽象语法树，基于AST可以做优化或者代码生成工作。

调试查看得到的AST，/src/compiler/parser/index.js，结构如下：

```
▼ root: Object
  ▶ attrs: [{...}]
  ▶ attrsList: [{...}]
  ▶ attrsMap: {id: "demo"}
  ▼ children: Array(3)
    ▶ 0: {type: 1, tag: "h1", attrsList: Array(0), attrsMap: {...},
    ▶ 1: {type: 3, text: " ", start: 37, end: 42}
    ▶ 2: {type: 1, tag: "p", attrsList: Array(0), attrsMap: {...},
      length: 3
    ▶ __proto__: Array(0)
  end: 65
  parent: undefined
  plain: false
  ▶ rawAttrsMap: {id: {...}}
  start: 0
  tag: "div"
  type: 1
```

解析器内部分了**HTML解析器**、**文本解析器**和**过滤器解析器**，最主要是HTML解析器

### 优化 - optimize

优化器的作用是在AST中找出静态子树并打上标记。静态子树是在AST中永远不变的节点，如纯文本节点。

标记静态子树的好处：

- 每次重新渲染，不需要为静态子树创建新节点
- 虚拟DOM中patch时，可以跳过静态子树

测试代码，06-2-compiler-optimize.html

代码实现，src/compiler/optimizer.js - optimize

标记结束

```
▼ ast: Object
  ▶ attrs: [{...}]
  ▶ attrsList: [{...}]
  ▶ attrsMap: {id: "demo"}
  ▶ children: (3) [{...}, {...}, {...}]
    end: 65
    parent: undefined
    plain: false
  ▶ rawAttrsMap: {id: {...}}
    start: 0
    static: false
    staticRoot: false
    tag: "div"
    type: 1
```

### 代码生成 - generate

将AST转换成渲染函数中的内容，即代码字符串。

generate方法生成渲染函数代码，**src/compiler/codegen/index.js**

生成的code长这样

```
`_c('div',{attrs:{"id":"demo"}},[
  _c('h1',[_v("Vue.js测试")]),
  _c('p',[_v(_s(foo))])
])`
```

### 典型指令的实现：v-if、v-for

着重观察几个结构性指令的解析过程

解析v-if: **parser/index.js**

processIf用于处理v-if解析

解析结果：

```

▶ attrsList: []
▶ attrsMap: {v-if: "foo"}
▶ children: [{...}]
end: 46
if: "foo"
▶ ifConditions: (2) [{...}, {...}]
▶ parent: {type: 1, tag: "div...
  plain: true
▶ rawAttrsMap: {v-if: {...}}
  start: 20
  tag: "h1"
  type: 1

```

代码生成, **codegen/index.js**

genIfConditions等用于生成条件语句相关代码

生成结果:

```

"with(this){return _c('div',{attrs:{id:"demo"}},[
  (foo) ? _c('h1',[_v(_s(foo))]) : _c('h1',[_v("no title")]),
  _v(" "),_c('abc')],1)}"

```

解析v-for: **parser/index.js**

processFor用于处理v-for指令

解析结果: v-for="item in items" for:'items' alias:'item'

```

▼ el:
  type: 1
  tag: "b"
  ▶ attrsList: [{...}]
  ▶ attrsMap: {v-for: "s in arr", :key: "s"}
  ▶ rawAttrsMap: {v-for: {...}, :key: {...}}
  ▶ parent: {type: 1, tag: "div", attrsList: Arr...
  ▶ children: []
  start: 129
  end: 158
  for: "arr"
  alias: "s"
  ▶ __proto__: Object
  exp: "s in arr"
  ▶ res: {for: "arr", alias: "s"}
  this: undefined
  Return value: undefined

```

代码生成, `src\compiler\codegen\index.js`:

genFor用于生成相应代码

生成结果

```

"with(this){return _c('div',{attrs:{"id":"demo"}},[_m(0),_v(" "), (foo)?_c('p',
[_v(_s(foo))]):_e(),_v(" ")
,_l((arr),function(s){return _c('b',{key:s},[_v(_s(s))])})
,_v(" ") ,_c('comp')],2)}"

```

v-if, v-for这些指令只能在编译器阶段处理, 如果我们要在render函数处理条件或循环只能使用if和for

```

Vue.component('comp', {
  props: ['foo'],
  render(h) { // 渲染内容跟foo的值挂钩, 只能用if语句
    if (this.foo==='foo') {
      return h('div', 'foo')
    }
    return h('div', 'bar')
  }
})

```

```
(function anonymous(
) {
with(this){return _c('div',{attrs:{"id":"demo"}},[_m(0),_v(" "), (foo)?_c('p',
[_v(_s(foo))]):_e(),_v(" "),_c('comp')],1)}
})
```

## 组件化机制

### 组件声明

#### Vue.component()

initAssetRegisters(Vue) `src/core/global-api/assets.js`

组件注册使用extend方法将配置转换为构造函数并添加到components选项

### 组件实例创建及挂载

观察生成的渲染函数

```
"with(this){return _c('div',{attrs:{"id":"demo"}},[
  _c('h1',[_v("虚拟DOM")]),_v(" "),
  _c('p',[_v(_s(foo))]),_v(" "),
  _c('comp') // 对于组件的处理并无特殊之处
],1)}"
```

### 整体流程

首先创建的是根组件，首次\_render()时，会得到整棵树的VNode结构

整体流程：new Vue() => \$mount() => vm.\_render() => createElement() => createComponent()

=> patch => createElm => createComponent()

### 创建自定义组件VNode

\_createElement `src\core\vdom\create-element.js`

\_createElement实际执行VNode创建的函数，由于传入tag是非保留标签，因此判定为自定义组件通过createComponent去创建

createComponent `src\core\vdom\create-component.js`

创建组件VNode，保存了上一步处理得到的组件构造函数，props，事件等

注意组件钩子安装和组件tag指定规则

### 创建自定义组件实例

根组件执行更新函数时，会递归创建子元素和子组件，入口createElm

### createEle() core/vdom/patch.js line751

首次执行\_update()时，patch()会通过createEle()创建根元素，子元素创建研究从这里开始

### createComponent core/vdom/patch.js line144

自定义组件创建

```
// 组件实例创建、挂载
if (isDef(i = i.hook) && isDef(i = i.init)) {
  i(vnode, false /* hydrating */)
}

if (isDef(vnode.componentInstance)) {
  // 元素引用指定vnode.elm, 元素属性创建等
  initComponent(vnode, insertedVnodeQueue)
  // 插入到父元素
  insert(parentElm, vnode.elm, refElm)
  if (isTrue(isReactivated)) {
    reactivateComponent(vnode, insertedVnodeQueue, parentElm, refElm)
  }
  return true
}
```

结论：

- 组件创建顺序自上而下
- 组件挂载顺序自下而上

## 总结

Vue源码学习使我们能够深入理解原理，解答很多开发中的疑惑，规避很多潜在的错误，写出更好的代码。学习大神的代码，能够学习编程思想，设计模式，训练基本功，提升内力。

## 作业

- 事件处理
  - 原生事件
  - 自定义事件
- 双向绑定
  - 思路：编译结果：赋值、事件监听

尝试去看源码，解答你的疑惑



# 预告

---

Vue SSR

课上问题总结：

1.diff在比较中直接操作dom吗？

有变化就直接操作dom，dom操作本身不是宏操作是微任务，这些操作是在浏览器刷新之前操作都结束了，可以立刻用微任务方式拿到dom值

2.直接操作dom是批量更新吗？

是的

3.双指针比较？

vue做了假设收尾的判断，期望减少循环次数，在循环中会有四个指针，他们向中间去移动

4.除了第一次打补丁是直接删除追加vdom之外，第二次之后都是操作旧vdom树是吗？

是的。之前有删除情况是刚开始时两颗树级别，可能是因为一开始某颗树不存在导致的删除这种情况，在新旧比较的时候也可能由于某个值变化导致节点被删掉。

2.重新new Vue() 和原始实例Vue 的依赖收集怎么联系起来的？为什么新new Vue({data}) 数据修改，能和根实例Vue的依赖联系起来？

每个组件里都可能有data，意味着每一个组件在创建实例的时候其实都做过一次对于data的响应式的过程，响应式过程可能不是根实例那一次，实例的时候其实都做过一次对于data响应式的过程，不是根实例的那一次，所以在组件树的过程有很多次对于data的初始化以及响应式，new Vue代表只是根实例，但是不代表说没有其他相同的过程，因为其他还有很多子组件也执行了类似的过程