



## 复习

- 异步更新：批量异步执行组件更新

dep.notify() => watcher.update() => queueWatcher() => nextTick() => timerFunc()

- core/observer/index.js reactiveSetter() 通知更新
- watcher.js update() 入队
- core\observer\scheduler.js 加入异步任务
- core\util\next-tick.js 加入回调，启动任务队列
- timerFunc() 异步执行任务

- 虚拟DOM：利用patching算法转换虚拟DOM为DOM

watcher.run() => updateComponent() => \_render() => \_update() => vm.\_\_patch\_\_() => patch()

- watcher.js run()
- core/instance/lifecycle.js mountComponent() updateComponent()
- core/instance/render.js \_render()
- core/instance/lifecycle.js \_update()
- platforms\web\runtime\index.js \_\_patch\_\_
- core/vdom/patch.js patch()

```
<div id="demo">
  <h1>虚拟DOM</h1>
  <p>{{foo}}</p>
</div>
<script>
  // 创建实例
  const app = new Vue({
    el: '#demo',
    data: { foo: 'foo' },
    mounted() {
      setTimeout(() => {
        this.foo = 'fooooo'
      }, 1000);
    }
  });
</script>
```

## 作业

- 属性相关dom操作：将属性相关dom操作按hooks归类，在patchVnode时一起执行

```
// 定义钩子数组
const hooks = ['create', 'activate', 'update', 'remove', 'destroy']
export function createPatchFunction (backend) {
  // 平台特别节点操作、属性更新对象
  const { modules, nodeOps } = backend
  for (i = 0; i < hooks.length; ++i) {
    // 指定到cbs对象上: cbs.create = []
    cbs[hooks[i]] = []
    for (j = 0; j < modules.length; ++j) {
      if (isDef(modules[j][hooks[i]])) {
        // 添加到相应数组中:
        // cbs.create = [fn1,fn2,...]
        // cbs.update = [fn1,fn2,...]
        cbs[hooks[i]].push(modules[j][hooks[i]])
      }
    }
  }
}

function patchVnode (...) {
  if (isDef(data) && isPatchable(vnode)) {
    // 执行默认的钩子
    for (i = 0; i < cbs.update.length; ++i) cbs.update[i](oldvnode, vnode)
    // 执行用户定义的钩子
    if (isDef(i = data.hook) && isDef(i = i.update)) i(oldvnode, vnode)
  }
}
}
```

- 组件化

测试代码

```
<div id="demo">
  <h1>Vue组件化机制</h1>
  <comp></comp>
</div>

<script>
  vue.component('comp', {
    template: '<div>I am comp</div>'
  })
</script>
```

组件声明

Vue.component()或者components选项

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

```
export function initAssetRegisters (Vue: GlobalAPI) {
  // 'component', 'directive', 'filter'
  ASSET_TYPES.forEach(type => {
    // vue['component']
    vue[type] = function (
```

```

    id: string,
    definition: Function | Object
  ): Function | Object | void {
    // 组件注册
    if (type === 'component' && isPlainObject(definition)) {
      definition.name = definition.name || id
      // 使用extend方法，将传入组件配置转换为构造函数VueComponent
      definition = this.options._base.extend(definition)
    }
    // vue.options['components']['comp'] = VueComponent
    this.options[type + 's'][id] = definition
    return definition
  }
}
})
}

```

创建根组件：

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

\_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

```

// 获取tag对应的组件构造函数
else if ((!data || !data.pre) && isDef(Ctor = resolveAsset(context.$options,
'components', tag))) {
  // 使用createComponent创建vnode
  vnode = createComponent(Ctor, data, context, children, tag)
}

```

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

创建自定义组件实例

createEle() core\vdom\patch.js

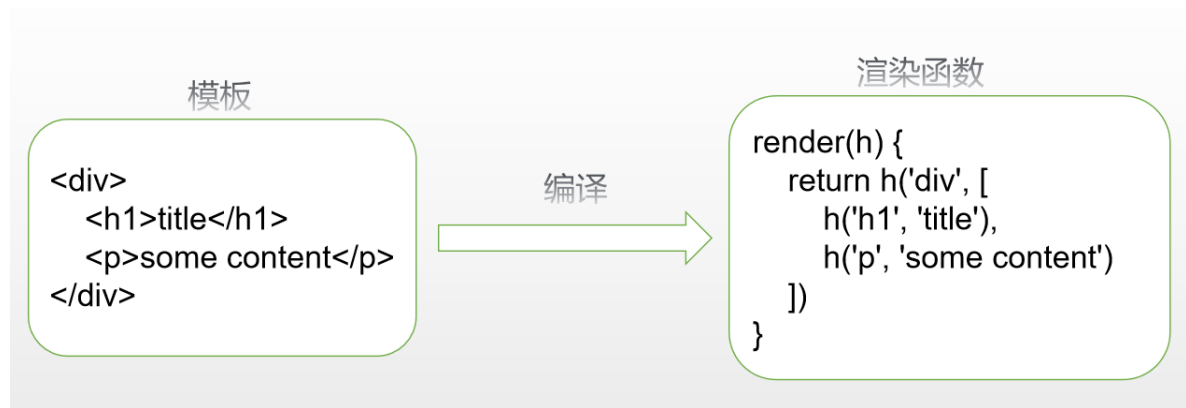
首次执行\_update()时，patch()会通过createEle()创建根元素，子元素创建研究从这里开始

createComponent core\vdom\patch.js

自定义组件创建

## 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)**



## 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

带编译器的版本中，可以使用template或el的方式声明模板

```
<div id="demo">
  <h1>Vue模板编译</h1>
  <p>{{foo}}</p>
  <comp></comp>
</div>
<script>
  vue.component('comp', {
    template: '<div>I am comp</div>'
  })
  // 创建实例
  const app = new Vue({
    el: '#demo',
    data: {foo: 'foo'}
  });
  // 输出render函数
  console.log(app.$options.render);
</script>
```

```
(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',{attrs:{"foo":"foo","bar":foo}})],1)}
})
"with(this){return _c('div',{attrs:{"id":"demo"}},[_m(0),_v(" "),_c('p',
[_v(_s(foo))]),_v(" "),_c('comp',{attrs:{"foo":"foo","bar":foo}})],1)}"
new Function(str)
```

\_c 返回vnode, createElement

\_v 创建文本节点

\_s 格式化函数

其他helpers: core/instance/render-helper/index

## 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

```
const { render, staticRenderFns } = compileToFunctions(template, {}, this)
```

### 编译过程

src\compiler\index.js

```
export const createCompiler = createCompilerCreator(function baseCompile (
  template: string,
  options: CompilerOptions
): CompiledResult {
  // 解析模板parse
  const ast = parse(template.trim(), options)
  if (options.optimize !== false) {
    optimize(ast, options) // 优化optimize
  }
  // 代码生成generate
  const code = generate(ast, options)
  return {
    ast,
    render: code.render,
    staticRenderFns: code.staticRenderFns
  }
})
```

## 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AST, 只有将模板解析成AST后, 才能基于它做优化或者生成代码字符串。

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，核心算法说明：

```

//src/compiler/parser/index.js
parseHTML(template, {
  start(tag, attrs, unary){}, // 遇到开始标签的处理
  end() {}, // 遇到结束标签的处理
  chars(text){}, // 遇到文本标签的处理
  comment(text){} // 遇到注释标签的处理
})

```

## 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

```

<!--要出现嵌套关系-->
<h1>Vue<span>模板编译</span></h1>

```

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

### v-if、v-for

着重观察几个结构性指令的解析过程

```
<p v-if="foo">{{foo}}</p>
```

解析v-if: parser/index.js

代码生成, codegen/index.js

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

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

```

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)}"

```

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js的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')
  }
})

```

## 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

- 尝试解答双向绑定实现原理
- 尝试解答事件处理机制



- 继续刷源码，解答开发中的疑惑，写出自己的理解，给大家分享一下更好

## 预告

---

Vue项目最佳实践

