

# Vue源码剖析01

---

## 作业

---

## 目标

---

- 环境搭建
- 理顺学习流程
- vue初始化过程
- 深入数据响应式

## 知识点

---

### 获取vue

项目地址: <https://github.com/vuejs/vue>

迁出项目: `git clone https://github.com/vuejs/vue.git`

当前版本号: 2.6.10

### 文件结构



源码目录



## 调试环境搭建

安装依赖: `npm i`

安装rollup: `npm i -g rollup`

修改dev脚本, 添加sourcemap, package.json

```
"dev": "rollup -w -c scripts/config.js --sourcemap --environment TARGET:web-full-dev",
```

运行开发命令: `npm run dev`

引入前面创建的vue.js, samples/commits/index.html

开课吧web全栈架构师

```
<script src="../../dist/vue.js"></script>
```

术语解释：

- runtime：仅包含运行时，不包含编译器
- common：cjs规范，用于webpack1
- esm：ES模块，用于webpack2+
- umd：universal module definition，兼容cjs和amd，用于浏览器

## 入口

dev脚本中 `-c scripts/config.js` 指明配置文件所在

参数 `TARGET:web-full-dev` 指明输出文件配置项，line:123

```
// Runtime+compiler development build (Browser)
{
  'web-full-dev': {
    entry: resolve('web/entry-runtime-with-compiler.js'), // 入口
    dest: resolve('dist/vue.js'), // 目标文件
    format: 'umd', // 输出规范
    env: 'development',
    alias: { he: './entity-decoder' },
    banner,
  },
}
```

## 初始化流程

入口 `platforms/web/entry-runtime-with-compiler.js`

扩展默认\$mount方法：处理template或el选项

`platforms/web/runtime/index.js`

安装web平台特有指令和组件

定义\_\_patch\_\_：补丁函数，执行patching算法进行更新

定义\$mount：挂载vue实例到指定宿主元素（获得dom并替换宿主元素）

`core/index.js`

初始化全局api

具体如下：

```

Vue.set = set
Vue.delete = del
Vue.nextTick = nextTick
initUse(Vue) // 实现Vue.use函数
initMixin(Vue) // 实现Vue.mixin函数
initExtend(Vue) // 实现Vue.extend函数
initAssetRegisters(Vue) // 注册实现Vue.component/directive/filter

```

## core/instance/index.js

Vue构造函数定义

定义Vue实例API

```

function Vue (options) {
  // 构造函数仅执行了_init
  this._init(options)
}

initMixin(Vue) // 实现init函数
stateMixin(Vue) // 状态相关api $data,$props,$set,$delete,$watch
eventsMixin(Vue) // 事件相关api $on,$once,$off,$emit
lifecycleMixin(Vue) // 生命周期api _update,$forceUpdate,$destroy
renderMixin(Vue) // 渲染api _render,$nextTick

```

## core/instance/init.js

创建组件实例，初始化其数据、属性、事件等

```

initLifecycle(vm) // $parent,$root,$children,$refs
initEvents(vm) // 处理父组件传递的事件和回调
initRender(vm) // $slots,$scopedSlots,_c,$createElement
callHook(vm, 'beforeCreate')
initInjections(vm) // 获取注入数据
initState(vm) // 初始化props, methods, data, computed, watch
initProvide(vm) // 提供数据注入
callHook(vm, 'created')

```

## mountComponent core/instance/lifecycle.js

执行挂载，获取vdom并转换为dom

## render() src\core\instance\render.js

渲染组件，获取vdom

## update() src\core\instance\lifecycle.js

执行更新，将传入vdom转换为dom，初始化时执行的是dom创建操作

## 总体流程:

`new Vue() => _init() => $mount() => _render() => _update()`

测试代码: `examples\test\01-init.html`

## 数据响应式

Vue一大特点是数据响应式，数据的变化会作用于UI而不用进行DOM操作。原理上来讲，是利用了JS语言特性[Object.defineProperty\(\)](#)，通过定义对象属性setter方法拦截对象属性变更，从而将数值的变化转换为UI的变化。

具体实现是在Vue初始化时，会调用initState，它会初始化data，props等，这里着重关注data初始化，

## 整体流程

**initState (vm: Component) src\core\instance\state.js**

初始化数据，包括props、methods、data、computed和watch

**initData核心代码是将data数据响应化**

```
function initData (vm: Component) {  
  // 执行数据响应化  
  observe(data, true /* asRootData */)  
}
```

**core/observer/index.js**

observe方法返回一个Observer实例

**core/observer/index.js**

Observer对象根据数据类型执行对应的响应化操作

defineReactive定义对象属性的getter/setter，getter负责添加依赖，setter负责通知更新

**core/observer/dep.js**

Dep负责管理一组Watcher，包括watcher实例的增删及通知更新

## Watcher

Watcher解析一个表达式并收集依赖，当数值变化时触发回调函数，常用于\$watch API和指令中。

每个组件也会有对应的Watcher，数值变化会触发其update函数导致重新渲染

```
export default class watcher {
  constructor () {}
  get () {}
  addDep (dep: Dep) {}
  update () {}
}
```

相关API: `$watcher`

测试代码examples\test\02-1-reactive.html

## 数组响应化

数组数据变化的侦测跟对象不同，我们操作数组通常使用push、pop、splice等方法，此时没有办法得知数据变化。所以vue中采取的策略是拦截这些方法并通知dep。

src\core\observer\array.js

为数组原型中的7个可以改变内容的方法定义拦截器

Observer中覆盖数组原型

```
if (Array.isArray(value)) {
  // 替换数组原型
  protoAugment(value, arrayMethods) // value.__proto__ = arrayMethods
  this.observeArray(value)
}
```

测试代码examples\test\02-2-reactive-arr.html

相关API: `vue.set()/delete()`

## 作业

1. 理出整体流程思维导图
2. 自己尝试编写测试案例调试
3. 自己研究一下Vue.set/delete/\$watch等API
4. 尝试看看vue异步更新是如何实现的

