makeMap
initGlobalAPI
createCompiler
Vue
vue constructor
initMixin
Vue.prototype._init
stateMixin
$data
$props
$set
$delete
$watch
eventsMixin
$on
$once
$off
$emit
lifecycleMixin
Vue.prototype._update
Vue.prototype.$forceUpdate
Vue.prototype.$destroy
renderMixin
Vue.prototype.$nextTick
Vue.prototype._render
Vue.prototype.__patch__
重要
Vue.prototype.$mount
重要
Vue.prototype._render
生成 VNode
updateComponent
vnode = vm._render()
vm._update(vnode, hydrating)
new Vue
初始化
this._init
根据是否是组件选择不同处理方式合并属性
if (options && options._isComponent)
yes
initInternalComponent(vm, options);
no
mergeOptions
initProxy
set
has
get
vm._renderProxy = new Proxy(vm, handlers);
initLifecycle
initEvents
updateComponentListeners
updateListeners
重要
initRender
vm._c
vm.$createElement
callHook(vm, 'beforeCreate');
initInjections
initState
重要
vm._watchers = []
initProps
isRoot
toggleObserving(false)
vm._props = {}
loop
defineReactive$$1
observe(value)
proxy(vm, "_props", key)
toggleObserving(true)
initMethods
vm[key] = typeof methods[key] !== 'function' ? noop : bind(methods[key], vm);
initData
vm.$options.data
vm._data = getData(vm.$options.data)
while (Object.keys(vm._data).length--)
proxy(vm, "_data", key);
observe(data, true)
initComputed
initWatch
initProvide
callHook(vm, 'created');
vm.$mount
重要
if (vm.$el) {
vm.$el.__vue__ = vm;
}
vm.__patch__
先将构造函数中的 $mount 存储在 mount 上
重新定义了一次 $mount
if (!$options.render)
if (!$options.template)
template = getOuterHTML(el)
子主题
if (template)
compileToFunctions
ref
render
staticRenderFns
赋值给 this.$options
this.$options
赋值
这一部分主要是编译
return mount.call(this, el, hydrating)
真正开始执行初始定义的$mount
渲染Vnode
callHook(vm, 'beforeMount');
updateComponent
new Watcher()
callHook(vm, 'beforeUpdate');
if (vm.$vnode == null) {
vm._isMounted = true;
callHook(vm, 'mounted');
}
return vm
callHook(vm, 'mounted');
这里主要是挂载
渲染Vnode方法
mountComponent(this, el, hydrating)
expOrFn就是updateComponent
创建监听者
Watcher
vm._watchers.push(this)
if (isRenderWatcher) {
vm._watcher = this;
}
this.getter = expOrFn
this.value = this.lazy
? undefined
: this.get();
get
pushTarget(this);
value = this.getter.call(vm, vm);
popTarget();
this.cleanupDeps()
addDep(dep)
!this.newDepIds.has(id) && !this.depIds.has(id)
dep.addSub
cleanupDeps
update
this.sync
true
this.run
false
queueWatcher(this);
nextTick(flushSchedulerQueue);
调用 this.run()
run
var value = this.get
id
newDeps
depIds
newDepIds
function observe
new Observer(value)
对 props 设置观察者模式
这里设置的是对vm._data的代理
对 props 设置观察者模式
Observer
实例化 Observer
value
dep = new Dep()
vmCount
def(value, '__ob__', this);
重要
将 observe 实例绑定到 value 对象的 __ob__ 属性上
Array.isArray(value)
true
if (hasProto) {
protoAugment(value, arrayMethods);
} else {
copyAugment(value, arrayMethods, arrayKeys);
}
this.observeArray(value);
待研究
false
this.walk(value);
Observer.prototype.walk
defineReactive$$1(obj, keys);
new Dep()
Object.defineProperty
重要
get: function reactiveGetter () {
var value = getter ? getter.call(obj) : val;
if (Dep.target) {
dep.depend();
if (childOb) {
childOb.dep.depend();
if (Array.isArray(value)) {
dependArray(value);
}
}
}
return value
}
重要
dep.depend
set: function reactiveSetter (newVal) {
var value = getter ? getter.call(obj) : val;
if (newVal === value || (newVal !== newVal && value !== value)) {
return
}
if (setter) {
setter.call(obj, newVal);
} else {
val = newVal;
}
childOb = !shallow && observe(newVal);
dep.notify();
}
get
set
Observer.prototype.observeArray
this.msg = 'ffffff'
修改data值
触发观察者的 set 方法
this.subs 和 this.id
判断 Dep上的target是否存在
Dep
addSub
removeSub
depend
notify
target
是当前的 watcher 实例
addDep()
调用target 的addDep
通过pushTarget将 watcher 实例
挂载到 Dep target 上
遍历调用this.subs下面的watcher
实例的update方法
createCompiler
template模板解析
调用此函数产生 render 函数
baseCompile
var ast = parse(template.trim(), options);
function parse(template, options)
parseHTML
return root
var code = generate(ast, options);
render: "with(this){return _c('div',{attrs:{"id":"app"}},[_c('p',[_v("我啥也不是")])]),_v(" "),_c('button',{on:{"click":toggle}},[_v("toggle")])])}"
staticRenderFns: []
return {
ast: ast,
render: code.render,
staticRenderFns: code.staticRenderFns
}
将 string 转换成 函数
compile
主要就是合并属性
return {
compile: compile,
compileToFunctions: createCompileToFunctionFn(compile)
}
createCompileToFunctionFn
将code转化成了function
compileToFunctions
render : function
staticRenderFns : function
返回的就是 compileToFunctions
这里通过闭包，对解析函数做了多层的封装，使用会更方便