Vue2源码解析（2/2）

1. 课程目标

掌握Vue2.6（目前2.X最高版本）的核心源码；

2. 课程大纲

- Vue组件化；

3. Vue组件化

Vue.js的一个核心思想是组件化。所谓组件化，就是把页面拆分成多个组件 (component)，每个组件依赖的 CSS、JavaScript、模板、图片等资源放在一起开发和维护。组件是资源独立的，组件在系统内部可复用，组件和组件之间可以嵌套，接下来我们会从源码的角度来分析 Vue 的组件内部是如何工作的。

接下来我们会用 Vue-cli 初始化的代码为例，来分析一下 Vue 组件初始化的一个过程。

```
import Vue from 'vue'
import App from './App.vue'

var app = new Vue({
  el: '#app',
  // 这里的 h 是 createElement 方法
  render: h => h(App)
})
```

这段代码相信很多同学都很熟悉，它和我们上一章相同的点也是通过 render 函数去渲染的，不同的这次通过 createElement 传的参数是一个组件而不是一个原生的标签，接下来我们就开始分析这一过程中发生了什么。

3.1. CreateComponent

上节课中，我们在createElement 的实现中讲过，它会调用 _createElement 方法，其中有一段逻辑是对参数 tag 的判断，如果是一个普通的 html 标签，会实例化一个普通的VNode节点，否则通过 createComponent 方法创建一个组件 VNode。

- src/code/vdom/create-element.js

```
if (typeof tag === 'string') {
  let Ctor
  ns = (context.$vnode && context.$vnode.ns) || config.getTagNamespace(tag)
  if (config.isReservedTag(tag)) {
    // platform built-in elements
    vnode = new VNode(
      config.parsePlatformTagName(tag), data, children,
      undefined, undefined, context
    )
  } else if (isDef(Ctor = resolveAsset(context.$options, 'components', tag))) {
    // component
    vnode = createComponent(Ctor, data, context, children, tag)
```

```
  } else {
    // unknown or unlisted namespaced elements
    // check at runtime because it may get assigned a namespace when its
    // parent normalizes children
    vnode = new VNode(
      tag, data, children,
      undefined, undefined, context
    )
  }
} else {
  // direct component options / constructor
  vnode = createComponent(tag, data, context, children)
}
```

在render函数中，我们传入的是 `render: h => h(App)` ，是一个Component组件类型，会执行

```
vnode = createComponent(Ctor, data, context, children, tag)
```

它定义在 `src/core/vdom/create-component.js` 文件中

```
export function createComponent (
  Ctor: Class<Component> | Function | Object | void,
  data: ?VNodeData,
  context: Component,
  children: ?Array<VNode>,
  tag?: string
): VNode | Array<VNode> | void {
  if (isUndef(Ctor)) {
    return
  }

  const baseCtor = context.$options._base

  // plain options object: turn it into a constructor
  if (isObject(Ctor)) {
    Ctor = baseCtor.extend(Ctor)
  }

  // if at this stage it's not a constructor or an async component factory,
  // reject.
  if (typeof Ctor !== 'function') {
    if (process.env.NODE_ENV !== 'production') {
      warn(`Invalid Component definition: ${String(Ctor)}`, context)
    }
    return
  }

  // async component
  let asyncFactory
  if (isUndef(Ctor.cid)) {
    asyncFactory = Ctor
    Ctor = resolveAsyncComponent(asyncFactory, baseCtor, context)
    if (Ctor === undefined) {
      // return a placeholder node for async component, which is rendered
      // as a comment node but preserves all the raw information for the node.
      // the information will be used for async server-rendering and hydration.
      return createAsyncPlaceholder(
```

```
      asyncFactory,
      data,
      context,
      children,
      tag
    )
  }
}

data = data || {}

// resolve constructor options in case global mixins are applied after
// component constructor creation
resolveConstructorOptions(Ctor)

// transform component v-model data into props & events
if (isDef(data.model)) {
  transformModel(Ctor.options, data)
}

// extract props
const propsData = extractPropsFromVNodeData(data, Ctor, tag)

// functional component
if (isTrue(Ctor.options.functional)) {
  return createFunctionalComponent(Ctor, propsData, data, context, children)
}

// extract listeners, since these needs to be treated as
// child component listeners instead of DOM listeners
const listeners = data.on
// replace with listeners with .native modifier
// so it gets processed during parent component patch.
data.on = data.nativeOn

if (isTrue(Ctor.options.abstract)) {
  // abstract components do not keep anything
  // other than props & listeners & slot

  // work around flow
  const slot = data.slot
  data = {}
  if (slot) {
    data.slot = slot
  }
}

// install component management hooks onto the placeholder node
installComponentHooks(data)

// return a placeholder vnode
const name = Ctor.options.name || tag
const vnode = new VNode(
  `vue-component-${Ctor.cid}${name ? `-${name}` : ''}`,
  data, undefined, undefined, undefined, context,
  { Ctor, propsData, listeners, tag, children },
  asyncFactory
)
```

```
  // Weex specific: invoke recycle-list optimized @render function for
  // extracting cell-slot template.
  // https://github.com/Hanks10100/weex-native-directive/tree/master/component
  /* istanbul ignore if */
  if (__WEEX__ && isRecyclableComponent(vnode)) {
    return renderRecyclableComponentTemplate(vnode)
  }

  return vnode
}
```

内容相对较多，我们只看核心的部分： `构造子类构造函数` ， `安装组件钩子函数` 和 `实例化 VNode` 。

3.1.1. 构造子类构造函数

```
const baseCtor = context.$options._base

// plain options object: turn it into a constructor
if (isObject(Ctor)) {
  Ctor = baseCtor.extend(Ctor)
}
```

在开发业务组件的时候，我们经常是使用下列方式创建：

```
import HelloWorld from './components/HelloWorld'

export default {
  name: 'app',
  components: {
    HelloWorld
  }
}
```

这里 export 的是一个对象，所以 `createComponent` 里的代码逻辑会执行到 `baseCtor.extend(Ctor)` ，在这里 baseCtor 实际上就是 Vue，这个的定义是在最开始初始化 Vue 的阶段，在 `src/core/global-api/index.js` 中的 `initGlobalAPI` 里定义的：

```
// this is used to identify the "base" constructor to extend all plain-object
  // components with in Weex's multi-instance scenarios.
  Vue.options._base = Vue
```

但是我们会发现，这里定义的是 `Vue.options` ，而我们的 createComponent 取的是 `context.$options` ，实际上在 `src/core/instance/init.js` 里 Vue 原型上的 _init 函数中注入的这部分内容：

```
vm.$options = mergeOptions(
  resolveConstructorOptions(vm.constructor),
  options || {},
  vm
)
```

这样就把 Vue 上的一些 option 扩展到了 `vm.$options` 上，所以我们也就能通过 `vm.$options._base` 拿到 Vue 这个构造函数了。mergeOptions后面详细讲解，现在只需要理解它的功能是把 Vue 构造函数的 options 和用户传入的 options 做一层合并，到 `vm.$options` 上。

在明确 `baseCtor` 指向了 Vue 之后，我们来看一下 Vue.extend 函数的定义，在 `src/core/global-api/extend.js` 中：

```
/**
  * Class inheritance
  */
 Vue.extend = function (extendOptions: Object): Function {
  extendOptions = extendOptions || {}
  const Super = this
  const SuperId = Super.cid
  const cachedCtors = extendOptions._Ctor || (extendOptions._Ctor = {})
  if (cachedCtors[SuperId]) {
   return cachedCtors[SuperId]
  }

  const name = extendOptions.name || Super.options.name
  if (process.env.NODE_ENV !== 'production' && name) {
   validateComponentName(name)
  }

  const Sub = function VueComponent (options) {
   this._init(options)
  }
  Sub.prototype = Object.create(Super.prototype)
  Sub.prototype.constructor = Sub
  Sub.cid = cid++
  Sub.options = mergeOptions(
   Super.options,
   extendOptions
  )
  Sub['super'] = Super

  // For props and computed properties, we define the proxy getters on
  // the Vue instances at extension time, on the extended prototype. This
  // avoids Object.defineProperty calls for each instance created.
  if (Sub.options.props) {
   initProps(Sub)
  }
  if (Sub.options.computed) {
   initComputed(Sub)
  }

  // allow further extension/mixin/plugin usage
  Sub.extend = Super.extend
  Sub.mixin = Super.mixin
  Sub.use = Super.use

  // create asset registers, so extended classes
  // can have their private assets too.
  ASSET_TYPES.forEach(function (type) {
   Sub[type] = Super[type]
  })
  // enable recursive self-lookup
  if (name) {
   Sub.options.components[name] = Sub
  }

  // keep a reference to the super options at extension time.
  // later at instantiation we can check if Super's options have
```

```
    // been updated.
    Sub.superOptions = Super.options
    Sub.extendOptions = extendOptions
    Sub.sealedOptions = extend({}, Sub.options)

    // cache constructor
    cachedCtors[SuperId] = Sub
    return Sub
  }
}
```

这里我们可以发现， `Vue.extend` 的作用就是通过原型继承的方式，把一个继承于Vue的构造器，Sub返回，然后对Sub这个对象本身扩展了一些属性，如扩展 options、添加全局 API 等；并且对配置中的 props 和 computed 做了初始化工作；最后对于这个 Sub 构造函数做了缓存，避免多次执行 Vue.extend 的时候对同一个子组件重复构造。

然后，当我们去实例化 Sub 的时候，就会执行 `this._init` 逻辑再次走到了 Vue 实例的初始化逻辑，实例化组件的方式后面说明。

```
const Sub = function VueComponent (options) {
  this._init(options)
}
```

3.1.2. 安装钩子函数

```
// install component management hooks onto the placeholder node
installComponentHooks(data
```

之前提到 Vue.js 使用的 Virtual DOM 参考的是开源库 snabbdom，它的一个特点是在 VNode 的 patch 流程中对外暴露了各种时机的钩子函数，方便我们做一些额外的事情，Vue.js 也是充分利用这一点，在初始化一个 Component 类型的 VNode 的过程中实现了几个钩子函数：

```
const componentVNodeHooks = {
  init (vnode: VNodeWithData, hydrating: boolean): ?boolean {
    if (
      vnode.componentInstance &&
      !vnode.componentInstance._isDestroyed &&
      vnode.data.keepAlive
    ) {
      // kept-alive components, treat as a patch
      const mountedNode: any = vnode // work around flow
      componentVNodeHooks.prepatch(mountedNode, mountedNode)
    } else {
      const child = vnode.componentInstance = createComponentInstanceForVnode(
        vnode,
        activeInstance
      )
      child.$mount(hydrating ? vnode.elm : undefined, hydrating)
    }
  },

  prepatch (oldVnode: MountedComponentVNode, vnode: MountedComponentVNode) {
    const options = vnode.componentOptions
    const child = vnode.componentInstance = oldVnode.componentInstance
    updateChildComponent(
      child,
      options.propsData, // updated props
```

```
      options.listeners, // updated listeners
      vnode, // new parent vnode
      options.children // new children
    )
  },

  insert (vnode: MountedComponentVNode) {
    const { context, componentInstance } = vnode
    if (!componentInstance._isMounted) {
      componentInstance._isMounted = true
      callHook(componentInstance, 'mounted')
    }
    if (vnode.data.keepAlive) {
      if (context._isMounted) {
        // vue-router#1212
        // During updates, a kept-alive component's child components may
        // change, so directly walking the tree here may call activated hooks
        // on incorrect children. Instead we push them into a queue which will
        // be processed after the whole patch process ended.
        queueActivatedComponent(componentInstance)
      } else {
        activateChildComponent(componentInstance, true /* direct */)
      }
    }
  },

  destroy (vnode: MountedComponentVNode) {
    const { componentInstance } = vnode
    if (!componentInstance._isDestroyed) {
      if (!vnode.data.keepAlive) {
        componentInstance.$destroy()
      } else {
        deactivateChildComponent(componentInstance, true /* direct */)
      }
    }
  }
}

const hooksToMerge = Object.keys(componentVNodeHooks)

function installComponentHooks (data: VNodeData) {
  const hooks = data.hook || (data.hook = {})
  for (let i = 0; i < hooksToMerge.length; i++) {
    const key = hooksToMerge[i]
    const existing = hooks[key]
    const toMerge = componentVNodeHooks[key]
    if (existing !== toMerge && !(existing && existing._merged)) {
      hooks[key] = existing ? mergeHook(toMerge, existing) : toMerge
    }
  }
}

function mergeHook (f1: any, f2: any): Function {
  const merged = (a, b) => {
    // flow complains about extra args which is why we use any
    f1(a, b)
    f2(a, b)
  }
```

```
  merged._merged = true
  return merged
}
```

整个 `installComponentHooks` 的过程就是把 `componentVNodeHooks` 的钩子函数合并到 `data.hook` 中，在 VNode 执行 patch 的过程中执行相关的钩子函数，具体的执行我们在后面 `patch` 过程中介绍。这里要注意的是合并策略，在合并过程中，如果某个时机的钩子已经存在 `data.hook` 中，那么通过执行 `mergeHook` 函数做合并，确保两个需要合并的钩子函数都执行。

3.1.3. 实例化VNode

```
const name = Ctor.options.name || tag
const vnode = new VNode(
  `vue-component-${Ctor.cid}${name ? `-${name}` : ''}`,
  data, undefined, undefined, undefined, context,
  { Ctor, propsData, listeners, tag, children },
  asyncFactory
)
return vnode
```

最后，我们会通过 `new VNode` 方式实例化一个VNode并返回，这里要注意，此时的Component VNode元素是没有children的，且前缀是 `vue-component` 。这块在后续的patch过程中会用到。

3.1.4. 总结

到这里，对组件我们完成了 `createComponent` 的处理，此时我们生成了组件的VNode，接下来按照正常的元素渲染，_render_已经处理完成，需要进行 `_update` ，也就是 `patch` 的逻辑。

3.2. patch

跟普通元素类似， `createComponent` 创建了组件 VNode，接下来会走到 v `m._update` ，执行 `vm.__patch__` 去把 VNode 转换成真正的 DOM 节点。我们对比普通元素，看下patch阶段对组件元素会有何种区别：

patch 的过程会调用 `createElm` 创建元素节点，回顾一下 createElm 的实现，它的定义在 `src/core/vdom/patch.js` 中：

```
function createElm (
  vnode,
  insertedVnodeQueue,
  parentElm,
  refElm,
  nested,
  ownerArray,
  index
) {
  // ...
  if (createComponent(vnode, insertedVnodeQueue, parentElm, refElm)) {
    return
  }
  // ...
}
```

3.2.1. createComponent

上节课有说过，对于非组件元素，我们在 `createComponent` 会返回false，接下来看下具体实现：

```
function createComponent (vnode, insertedVnodeQueue, parentElm, refElm) {
  let i = vnode.data
  if (isDef(i)) {
    const isReactivated = isDef(vnode.componentInstance) && i.keepAlive
    if (isDef(i = i.hook) && isDef(i = i.init)) {
      i(vnode, false /* hydrating */)
    }
    // after calling the init hook, if the vnode is a child component
    // it should've created a child instance and mounted it. the child
    // component also has set the placeholder vnode's elm.
    // in that case we can just return the element and be done.
    if (isDef(vnode.componentInstance)) {
      initComponent(vnode, insertedVnodeQueue)
      insert(parentElm, vnode.elm, refElm)
      if (isTrue(isReactivated)) {
        reactivateComponent(vnode, insertedVnodeQueue, parentElm, refElm)
      }
      return true
    }
  }
}
```

首先对vnode.data进行判断：

```
let i = vnode.data
if (isDef(i)) {
  // ...
  if (isDef(i = i.hook) && isDef(i = i.init)) {
    i(vnode, false /* hydrating */)
    // ...
  }
  // ..
}
```

如果 vnode 是一个组件 VNode，那么条件会满足，并且得到 i 就是 init 钩子函数，回顾上节我们在创建组件 VNode 的时候合并钩子函数中就包含 init 钩子函数，定义在 `src/core/vdom/create-component.js` 中：

```
init (vnode: VNodeWithData, hydrating: boolean): ?boolean {
  if (
    vnode.componentInstance &&
    !vnode.componentInstance._isDestroyed &&
    vnode.data.keepAlive
  ) {
    // kept-alive components, treat as a patch
    const mountedNode: any = vnode // work around flow
    componentVNodeHooks.prepatch(mountedNode, mountedNode)
  } else {
    const child = vnode.componentInstance = createComponentInstanceForVnode(
      vnode,
      activeInstance
    )
```

```
    child.$mount(hydrating ? vnode.elm : undefined, hydrating)
  }
},
```

我们先不考虑 keepAlive 的情况，它是通过 createComponentInstanceForVnode 创建一个 Vue 的实例，然后调用 $mount 方法挂载子组件， 先来看一下 createComponentInstanceForVnode 的实现：

```
export function createComponentInstanceForVnode (
  vnode: any, // we know it's MountedComponentVNode but flow doesn't
  parent: any, // activeInstance in lifecycle state
): Component {
  const options: InternalComponentOptions = {
    _isComponent: true,
    _parentVnode: vnode,
    parent
  }
  // check inline-template render functions
  const inlineTemplate = vnode.data.inlineTemplate
  if (isDef(inlineTemplate)) {
    options.render = inlineTemplate.render
    options.staticRenderFns = inlineTemplate.staticRenderFns
  }
  return new vnode.componentOptions.Ctor(options)
}
```

createComponentInstanceForVnode 函数构造的一个内部组件的参数，然后执行 new vnode.componentOptions.Ctor(options) 。这里的 vnode.componentOptions.Ctor 对应的就是子组件的构造函数，我们上一节分析了它实际上是继承于 Vue 的一个构造器 Sub，相当于 new Sub(options) 这里有几个关键参数要注意几个点，_isComponent 为 true 表示它是一个组件。

所以子组件的实例化实际上就是在这个时机执行的，并且它会执行实例的 _init 方法，这个过程有一些和之前不同的地方需要挑出来说，代码在 src/core/instance/init.js 中：

```
Vue.prototype._init = function (options?: Object) {
  const vm: Component = this
  // merge options
  if (options && options._isComponent) {
    // optimize internal component instantiation
    // since dynamic options merging is pretty slow, and none of the
    // internal component options needs special treatment.
    initInternalComponent(vm, options)
  } else {
    vm.$options = mergeOptions(
      resolveConstructorOptions(vm.constructor),
      options || {},
      vm
    )
  }
  // ...
  if (vm.$options.el) {
    vm.$mount(vm.$options.el)
  }
}
```

首先是合并 options 的过程有变化，_isComponent 为 true，所以走到了 initInternalComponent 过程：

```
export function initInternalComponent (vm: Component, options: InternalComponentOptions) {
  const opts = vm.$options = Object.create(vm.constructor.options)
  // doing this because it's faster than dynamic enumeration.
  const parentVnode = options._parentVnode
  opts.parent = options.parent
  opts._parentVnode = parentVnode

  const vnodeComponentOptions = parentVnode.componentOptions
  opts.propsData = vnodeComponentOptions.propsData
  opts._parentListeners = vnodeComponentOptions.listeners
  opts._renderChildren = vnodeComponentOptions.children
  opts._componentTag = vnodeComponentOptions.tag

  if (options.render) {
    opts.render = options.render
    opts.staticRenderFns = options.staticRenderFns
  }
}
```

> 这里着重注意下： `opts.parent = options.parent` 、 `opts._parentVnode = parentVnode`

最后， `_init` 执行：

```
if (vm.$options.el) {
   vm.$mount(vm.$options.el)
}
```

此时，回顾组件的init

```
init (vnode: VNodeWithData, hydrating: boolean): ?boolean {
  if (
    vnode.componentInstance &&
    !vnode.componentInstance._isDestroyed &&
    vnode.data.keepAlive
  ) {
    // kept-alive components, treat as a patch
    const mountedNode: any = vnode // work around flow
    componentVNodeHooks.prepatch(mountedNode, mountedNode)
  } else {
    const child = vnode.componentInstance = createComponentInstanceForVnode(
      vnode,
      activeInstance
    )
    child.$mount(hydrating ? vnode.elm : undefined, hydrating)
  }
},
```

会执行 `child.$mount(hydrating ? vnode.elm : undefined, hydrating)` 。这里 hydrating 为 true 一般是服务端渲染的情况，我们只考虑客户端渲染，所以这里 $mount 相当于执行 `child.$mount(undefined, false)` ，它最终会调用 mountComponent 方法，进而执行 vm._render() 方法：

```
Vue.prototype._render = function (): VNode {
  const vm: Component = this
  const { render, _parentVnode } = vm.$options
```

```
  // set parent vnode. this allows render functions to have access
  // to the data on the placeholder node.
  vm.$vnode = _parentVnode
  // render self
  let vnode
  try {
    vnode = render.call(vm._renderProxy, vm.$createElement)
  } catch (e) {
    // ...
  }
  // set parent
  vnode.parent = _parentVnode
  return vnode
}
```

这里的 `_parentVnode` 就是当前组件的父 VNode，而 render 函数生成的 vnode 当前组件的渲染 vnode，vnode 的 parent 指向了 `_parentVnode`，也就是 vm.$vnode，它们是一种父子的关系，在执行完 vm._render 生成 VNode 后，接下来就要执行 `vm._update` 去渲染 VNode 了。来看一下组件渲染的过程中有哪些需要注意的，`vm._update` 的定义在 `src/core/instance/lifecycle.js` 中：

```
export let activeInstance: any = null
Vue.prototype._update = function (vnode: VNode, hydrating?: boolean) {
  const vm: Component = this
  const prevEl = vm.$el
  const prevVnode = vm._vnode
  const prevActiveInstance = activeInstance
  activeInstance = vm
  vm._vnode = vnode
  // Vue.prototype.__patch__ is injected in entry points
  // based on the rendering backend used.
  if (!prevVnode) {
    // initial render
    vm.$el = vm.__patch__(vm.$el, vnode, hydrating, false /* removeOnly */)
  } else {
    // updates
    vm.$el = vm.__patch__(prevVnode, vnode)
  }
  activeInstance = prevActiveInstance
  // update __vue__ reference
  if (prevEl) {
    prevEl.__vue__ = null
  }
  if (vm.$el) {
    vm.$el.__vue__ = vm
  }
  // if parent is an HOC, update its $el as well
  if (vm.$vnode && vm.$parent && vm.$vnode === vm.$parent._vnode) {
    vm.$parent.$el = vm.$el
  }
  // updated hook is called by the scheduler to ensure that children are
  // updated in a parent's updated hook.
}
```

首先，`vm._vnode = vnode` 的逻辑，这个 vnode 是通过 vm._render() 返回的组件渲染 VNode，`vm._vnode` 和 `vm.$vnode` 的关系就是一种父子关系，用代码表达就是 `vm._vnode.parent === vm.$vnode`

```
export let activeInstance: any = null
Vue.prototype._update = function (vnode: VNode, hydrating?: boolean) {
  // ...
  const prevActiveInstance = activeInstance
  activeInstance = vm
  if (!prevVnode) {
    // initial render
    vm.$el = vm.__patch__(vm.$el, vnode, hydrating, false /* removeOnly */)
  } else {
    // updates
    vm.$el = vm.__patch__(prevVnode, vnode)
  }
  activeInstance = prevActiveInstance
  // ...
}
```

这个 activeInstance 作用就是保持当前上下文的 Vue 实例，它是在 lifecycle 模块的全局变量，定义是 `export let activeInstance: any = null`，并且在之前我们调用 `createComponentInstanceForVnode` 方法的时候从 lifecycle 模块获取，并且作为参数传入的。因为实际上 JavaScript 是一个单线程，Vue 整个初始化是一个深度遍历的过程，在实例化子组件的过程中，它需要知道当前上下文的 Vue 实例是什么，并把它作为子组件的父 Vue 实例。前面我们提到过对子组件的实例化过程先会调用 `initInternalComponent(vm, options)` 合并 options，把 parent 存储在 vm.$options 中，在 $mount 之前会调用 initLifecycle(vm) 方法：

```
export function initLifecycle (vm: Component) {
  const options = vm.$options

  // locate first non-abstract parent
  let parent = options.parent
  if (parent && !options.abstract) {
    while (parent.$options.abstract && parent.$parent) {
      parent = parent.$parent
    }
    parent.$children.push(vm)
  }

  vm.$parent = parent
  // ...
}
```

可以看到 vm.$parent 就是用来保留当前 vm 的父实例，并且通过 parent.$children.push(vm) 来把当前的 vm 存储到父实例的 $children 中。

在 vm._update 的过程中，把当前的 vm 赋值给 activeInstance，同时通过 const prevActiveInstance = activeInstance 用 prevActiveInstance 保留上一次的 activeInstance。实际上，prevActiveInstance 和当前的 vm 是一个父子关系，当一个 vm 实例完成它的所有子树的 patch 或者 update 过程后，activeInstance 会回到它的父实例，这样就完美地保证了 `createComponentInstanceForVnode` 整个深度遍历过程中，我们在实例化子组件的时候能传入当前子组件的父 Vue 实例，并在 _init 的过程中，通过 vm.$parent 把这个父子关系保留。

那么回到 _update，最后就是调用 __patch__ 渲染 VNode：

```
vm.$el = vm.__patch__(vm.$el, vnode, hydrating, false /* removeOnly */)

function patch (oldVnode, vnode, hydrating, removeOnly) {
  // ...
  let isInitialPatch = false
```

```
  const insertedVnodeQueue = []

  if (isUndef(oldVnode)) {
    // empty mount (likely as component), create new root element
    isInitialPatch = true
    createElm(vnode, insertedVnodeQueue)
  } else {
    // ...
  }
  // ...
}
```

之前分析过负责渲染成 DOM 的函数是 createElm，注意这里我们只传了 2 个参数，所以对应的 parentElm 是 undefined。我们再来看看它的定义：

```
function createElm (
  vnode,
  insertedVnodeQueue,
  parentElm,
  refElm,
  nested,
  ownerArray,
  index
) {
  // ...
  if (createComponent(vnode, insertedVnodeQueue, parentElm, refElm)) {
    return
  }

  const data = vnode.data
  const children = vnode.children
  const tag = vnode.tag
  if (isDef(tag)) {
    // ...

    vnode.elm = vnode.ns
      ? nodeOps.createElementNS(vnode.ns, tag)
      : nodeOps.createElement(tag, vnode)
    setScope(vnode)

    /* istanbul ignore if */
    if (__WEEX__) {
      // ...
    } else {
      createChildren(vnode, children, insertedVnodeQueue)
      if (isDef(data)) {
        invokeCreateHooks(vnode, insertedVnodeQueue)
      }
      insert(parentElm, vnode.elm, refElm)
    }

    // ...
  } else if (isTrue(vnode.isComment)) {
    vnode.elm = nodeOps.createComment(vnode.text)
    insert(parentElm, vnode.elm, refElm)
  } else {
    vnode.elm = nodeOps.createTextNode(vnode.text)
    insert(parentElm, vnode.elm, refElm)
```

```
  }
}
```

> 注意，这里我们传入的 vnode 是组件渲染的 vnode，也就是我们之前说的 vm._vnode，如果组件的根节点是个普通元素，那么 vm._vnode 也是普通的 vnode，这里 `createComponent(vnode, insertedVnodeQueue, parentElm, refElm)` 的返回值是 false。接下来的过程就和我们上一章一样了，先创建一个父节点占位符，然后再遍历所有子 VNode 递归调用 createElm，在遍历的过程中，如果遇到子 VNode 是一个组件的 VNode，则重复本节开始的过程，这样通过一个递归的方式就可以完整地构建了整个组件树。

由于我们这个时候传入的 `parentElm` 是空，所以对组件的插入，在 `createComponent` 有这么一段逻辑：

```
function createComponent (vnode, insertedVnodeQueue, parentElm, refElm) {
  let i = vnode.data
  if (isDef(i)) {
    // ....
    if (isDef(i = i.hook) && isDef(i = i.init)) {
      i(vnode, false /* hydrating */)
    }
    // ...
    if (isDef(vnode.componentInstance)) {
      initComponent(vnode, insertedVnodeQueue)
      insert(parentElm, vnode.elm, refElm)
      if (isTrue(isReactivated)) {
        reactivateComponent(vnode, insertedVnodeQueue, parentElm, refElm)
      }
      return true
    }
  }
}
```

在完成组件的整个 patch 过程后，最后执行 `insert(parentElm, vnode.elm, refElm)` 完成组件的 DOM 插入，如果组件 patch 过程中又创建了子组件，那么DOM 的插入顺序是先子后父。

至此，我们就讲完了一个组件VNode是如何创建、初始化、渲染的了。

3.3. 合并配置

我们知道，`new Vue` 的过程通常有 2 种场景，一种是外部我们的代码主动调用 `new Vue(options)` 的方式实例化一个 Vue 对象；另一种是我们上一节分析的组件过程中内部通过 `new Vue(options)` 实例化子组件

两者都会执行实例的 `_init(options)` 方法，它首先会执行一个 `merge options` 的逻辑，相关的代码在 `src/core/instance/init.js` 中：

```
Vue.prototype._init = function (options?: Object) {
  // merge options
  if (options && options._isComponent) {
    // optimize internal component instantiation
    // since dynamic options merging is pretty slow, and none of the
    // internal component options needs special treatment.
    initInternalComponent(vm, options)
  } else {
    vm.$options = mergeOptions(
```

```
      resolveConstructorOptions(vm.constructor),
      options || {},
      vm
    )
  }
  // ...
}
```

可以发现，options有两种类型的传参，接下来会对options进行分析：

首先，我们举个简单的例子：

```
import Vue from 'vue'

let childComp = {
  template: '<div>{{msg}}</div>',
  created() {
    console.log('child created')
  },
  mounted() {
    console.log('child mounted')
  },
  data() {
    return {
      msg: 'Hello Vue'
    }
  }
}

Vue.mixin({
  created() {
    console.log('parent created')
  }
})

let app = new Vue({
  el: '#app',
  render: h => h(childComp)
})
```

3.3.1. 外部调用场景

当执行 new Vue 的时候，在执行 `this._init(options)` 的时候，就会执行如下逻辑去合并 options：

```
vm.$options = mergeOptions(
  resolveConstructorOptions(vm.constructor),
  options || {},
  vm
)
```

通过调用 `mergeOptions` 方法来合并，它实际上就是把 `resolveConstructorOptions(vm.constructor)` 的返回值和 `options` 做合并， `resolveConstructorOptions` 的实现先不考虑，在我们这个场景下，它还是简单返回 `vm.constructor.options` ，相当于 Vue.options，那么这个值又是什么呢，其实在 initGlobalAPI(Vue) 的时候定义了这个值，在 `src/core/global-api/index.js` 中：

```
export function initGlobalAPI (Vue: GlobalAPI) {
  // ...
  Vue.options = Object.create(null)
  ASSET_TYPES.forEach(type => {
    Vue.options[type + 's'] = Object.create(null)
  })

  // this is used to identify the "base" constructor to extend all plain-object
  // components with in Weex's multi-instance scenarios.
  Vue.options._base = Vue

  extend(Vue.options.components, builtInComponents)
  // ...
}
```

首先通过 `Vue.options = Object.create(null)` 创建一个空对象，然后遍历 `ASSET_TYPES`，ASSET_TYPES 的定义在 `src/shared/constants.js` 中：

```
export const ASSET_TYPES = [
  'component',
  'directive',
  'filter'
]

// 遍历结果为：
Vue.options.components = {}
Vue.options.directives = {}
Vue.options.filters = {}
```

接着执行了 `Vue.options._base = Vue`；

最后通过 `extend(Vue.options.components, builtInComponents)` 把一些内置组件扩展到 `Vue.options.components` 上，Vue 的内置组件目前有 <keep-alive>、<transition> 和 <transition-group> 组件，这也就是为什么我们在其它组件中使用 <keep-alive> 组件不需要注册的原因，后续详细讲解。

回到 `mergeOptions` 这个函数，它的定义在 `src/core/util/options.js` 中：

```
/**
 * Merge two option objects into a new one.
 * Core utility used in both instantiation and inheritance.
 */
export function mergeOptions (
  parent: Object,
  child: Object,
  vm?: Component
): Object {
  if (process.env.NODE_ENV !== 'production') {
    checkComponents(child)
  }

  if (typeof child === 'function') {
    child = child.options
  }

  normalizeProps(child, vm)
  normalizeInject(child, vm)
```

```
  normalizeDirectives(child)

  // Apply extends and mixins on the child options,
  // but only if it is a raw options object that isn't
  // the result of another mergeOptions call.
  // Only merged options has the _base property.
  if (!child._base) {
    if (child.extends) {
      parent = mergeOptions(parent, child.extends, vm)
    }
    if (child.mixins) {
      for (let i = 0, l = child.mixins.length; i < l; i++) {
        parent = mergeOptions(parent, child.mixins[i], vm)
      }
    }
  }

  const options = {}
  let key
  for (key in parent) {
    mergeField(key)
  }
  for (key in child) {
    if (!hasOwn(parent, key)) {
      mergeField(key)
    }
  }
  function mergeField (key) {
    const strat = strats[key] || defaultStrat
    options[key] = strat(parent[key], child[key], vm, key)
  }
  return options
}
```

主要功能就是把 parent 和 child 这两个对象根据一些合并策略，合并成一个新对象并返回。比较核心的几步，先递归把 extends 和 mixins 合并到 parent 上，然后遍历 parent，调用 mergeField，然后再遍历 child，如果 key 不在 parent 的自身属性上，则调用 `mergeField`

其中， `mergeField` 函数，它对不同的 key 有着不同的合并策略。举例来说，对于生命周期函数，它的合并策略是这样的：

```
function mergeHook (
  parentVal: ?Array<Function>,
  childVal: ?Function | ?Array<Function>
): ?Array<Function> {
  return childVal
    ? parentVal
      ? parentVal.concat(childVal)
      : Array.isArray(childVal)
        ? childVal
        : [childVal]
    : parentVal
}
```

```
LIFECYCLE_HOOKS.forEach(hook => {
  strats[hook] = mergeHook
})
```

其中的 `LIFECYCLE_HOOKS` 的定义在 `src/shared/constants.js` 中：

```
export const LIFECYCLE_HOOKS = [
  'beforeCreate',
  'created',
  'beforeMount',
  'mounted',
  'beforeUpdate',
  'updated',
  'beforeDestroy',
  'destroyed',
  'activated',
  'deactivated',
  'errorCaptured'
]
```

这里定义了 Vue.js 所有的钩子函数名称，所以对于钩子函数，他们的合并策略都是 mergeHook 函数。

而 `nergeHook` 用了一个多层 3 元运算符，逻辑就是如果不存在 childVal ，就返回 parentVal；否则再判断是否存在 parentVal，如果存在就把 childVal 添加到 `parentVal` 后返回新数组；否则返回 `childVal` 的数组。所以回到 `mergeOptions` 函数，一旦 parent 和 child 都定义了相同的钩子函数，那么它们会把 2 个钩子函数合并成一个数组。

> 其他属性的内容在src/core/util/options

通过执行 `mergeField` 函数，把合并后的结果保存到 options 对象中，最终返回它。

因此，在我们当前这个 case 下，执行完如下合并后：

```
vm.$options = mergeOptions(
  resolveConstructorOptions(vm.constructor),
  options || {},
  vm
)
```

vm.$options 的值差不多是如下这样：

```
vm.$options = {
  components: { },
  created: [
    function created() {
      console.log('parent created')
    }
  ],
  directives: { },
  filters: { },
  _base: function Vue(options) {
    // ...
  },
  el: "#app",
  render: function (h) {
    //...
  }
}
```

3.3.2. 组件场景

组件的构造函数是通过 Vue.extend 继承自 Vue 的，先回顾一下这个过程，代码定义在 `src/core/global-api/extend.js` 中：

```
/**
 * Class inheritance
 */
Vue.extend = function (extendOptions: Object): Function {
  // ...
  Sub.options = mergeOptions(
    Super.options,
    extendOptions
  )

  // ...
  // keep a reference to the super options at extension time.
  // later at instantiation we can check if Super's options have
  // been updated.
  Sub.superOptions = Super.options
  Sub.extendOptions = extendOptions
  Sub.sealedOptions = extend({}, Sub.options)

  // ...
  return Sub
}
```

我们只保留关键逻辑，这里的 `extendOptions` 对应的就是前面定义的组件对象，它会和 Vue.options 合并到 Sub.opitons 中。

接下来我们再回忆一下子组件的初始化过程，代码定义在 `src/core/vdom/create-component.js` 中：

```
export function createComponentInstanceForVnode (
  vnode: any, // we know it's MountedComponentVNode but flow doesn't
  parent: any, // activeInstance in lifecycle state
): Component {
  const options: InternalComponentOptions = {
    _isComponent: true,
    _parentVnode: vnode,
    parent
  }
  // ...
  return new vnode.componentOptions.Ctor(options)
}
```

这里的 `vnode.componentOptions.Ctor` 就是指向 `Vue.extend` 的返回值 Sub， 所以 执行 n `ew vnode.componentOptions.Ctor(options)` 接着执行 this._init(options)，因为 `options._isComponent` 为 true，那么合并 options 的过程走到了 `initInternalComponent(vm, options)` 逻辑。在 `src/core/instance/init.js` 中：

```
export function initInternalComponent (vm: Component, options: InternalComponentOptions) {
  const opts = vm.$options = Object.create(vm.constructor.options)
  // doing this because it's faster than dynamic enumeration.
  const parentVnode = options._parentVnode
  opts.parent = options.parent
  opts._parentVnode = parentVnode
```

```
  const vnodeComponentOptions = parentVnode.componentOptions
  opts.propsData = vnodeComponentOptions.propsData
  opts._parentListeners = vnodeComponentOptions.listeners
  opts._renderChildren = vnodeComponentOptions.children
  opts._componentTag = vnodeComponentOptions.tag

  if (options.render) {
    opts.render = options.render
    opts.staticRenderFns = options.staticRenderFns
  }
}
```

initInternalComponent 方法首先执行 `const opts = vm.$options = Object.create(vm.constructor.options)`，这里的 vm.constructor 就是子组件的构造函数 Sub，相当于 `vm.$options = Object.create(Sub.options)` 。

接着又把实例化子组件传入的子组件父 VNode 实例 parentVnode、子组件的父 Vue 实例 parent 保存到 vm.$options 中，另外还保留了 parentVnode 配置中的如 propsData 等其它的属性。

这么看来，initInternalComponent 只是做了简单一层对象赋值，并不涉及到递归、合并策略等复杂逻辑。

因此，执行完成后

```
initInternalComponent(vm, options)
```

`vm.$options` 的值大致为:

```
vm.$options = {
  parent: Vue /*父Vue实例*/,
  propsData: undefined,
  _componentTag: undefined,
  _parentVnode: VNode /*父VNode实例*/,
  _renderChildren:undefined,
  __proto__: {
    components: { },
    directives: { },
    filters: { },
    _base: function Vue(options) {
        //...
    },
    _Ctor: {},
    created: [
      function created() {
        console.log('parent created')
      }, function created() {
        console.log('child created')
      }
    ],
    mounted: [
      function mounted() {
        console.log('child mounted')
      }
    ],
    data() {
      return {
        msg: 'Hello Vue'
      }
    },
```

```
    template: '<div>{{msg}}</div>'
  }
}
```

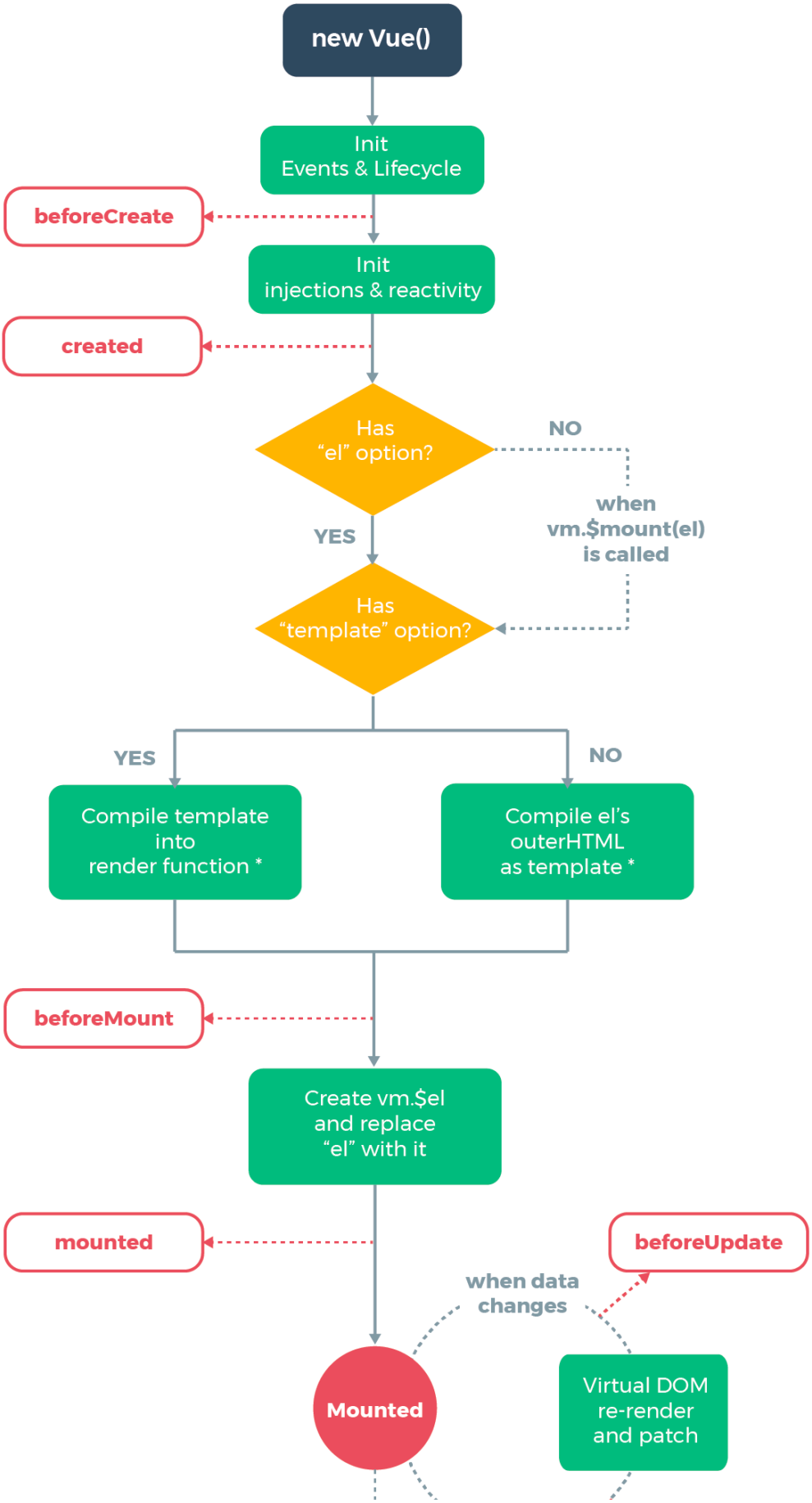
3.3.3. 总结

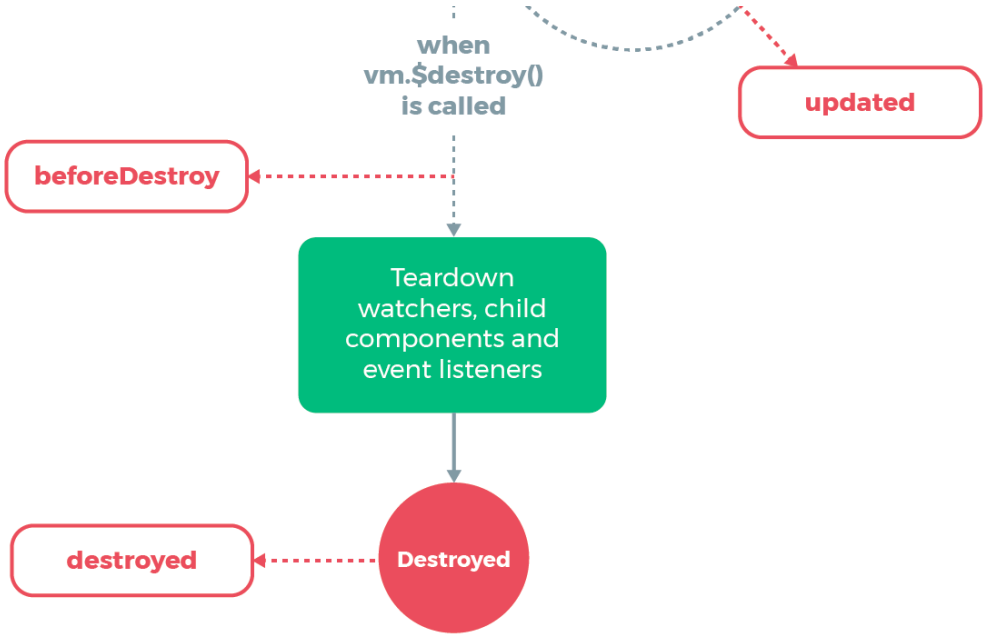
至此，Vue 初始化阶段对于 options 的合并过程就介绍完了，我们需要知道对于 options 的合并有 2 种方式，子组件初始化过程通过 `initInternalComponent` 方式要比外部初始化 Vue 通过 `mergeOptions` 的过程要快，合并完的结果保留在 `vm.$options` 中。

纵观一些库、框架的设计几乎都是类似的，自身定义了一些默认配置，同时又可以在初始化阶段传入一些定义配置，然后去 merge 默认配置，来达到定制化不同需求的目的，这个思路是值得借鉴的。

3.4. 生命周期

每个 Vue 实例在被创建之前都要经过一系列的初始化过程。例如需要设置数据监听、编译模板、挂载实例到 DOM、在数据变化时更新 DOM 等。同时在这个过程中也会运行一些叫做生命周期钩子的函数，给予用户机会在一些特定的场景下添加他们自己的代码。


new Vue()
Init
Events & Lifecycle
beforeCreate
Init
injections & reactivity
created
Has
“el” option?
NO
when
vm.$mount(el)
is called
YES
Has
“template” option?
YES
NO
Compile template
into
render function *
Compile el’s
outerHTML
as template *
beforeMount
Create vm.$el
and replace
“el” with it
mounted
beforeUpdate
when data
changes
Mounted
Virtual DOM
re-render
and patch



* template compilation is performed ahead-of-time if using a build step, e.g. single-file components

在实际项目开发过程中，会非常频繁地和 Vue 组件的生命周期打交道，接下来我们就从源码的角度来看一下这些生命周期的钩子函数是如何被执行的。

源码中最终执行生命周期的函数都是调用 callHook 方法，它的定义在 `src/core/instance/lifecycle` 中：

```
export function callHook (vm: Component, hook: string) {
  // #7573 disable dep collection when invoking lifecycle hooks
  pushTarget()
  const handlers = vm.$options[hook]
  if (handlers) {
    for (let i = 0, j = handlers.length; i < j; i++) {
      try {
        handlers[i].call(vm)
      } catch (e) {
        handleError(e, vm, `${hook} hook`)
      }
    }
  }
  if (vm._hasHookEvent) {
    vm.$emit('hook:' + hook)
  }
  popTarget()
}
```

callHooks根据传入的字符串 hook，去拿到 `vm.$options[hook]` 对应的回调函数数组，然后遍历执行，执行的时候把 vm 作为函数执行的上下文。

在上一节中，我们详细地介绍了 Vue.js 合并 options 的过程，各个阶段的生命周期的函数也被合并到 `vm.$options` 里，并且是一个数组。因此 `callhook` 函数的功能就是调用某个生命周期钩子注册的所有回调函数。

了解了生命周期的执行方式后，接下来我们会具体介绍每一个生命周期函数它的调用时机。

3.4.1. beforeCreate & created

`beforeCreate` 和 `created` 函数都是在实例化 Vue 的阶段，在 _init 方法中执行的，它的定义在 `src/core/instance/init.js` 中：

```
Vue.prototype._init = function (options?: Object) {
  // ...
  initLifecycle(vm)
  initEvents(vm)
  initRender(vm)
  callHook(vm, 'beforeCreate')
  initInjections(vm) // resolve injections before data/props
  initState(vm)
  initProvide(vm) // resolve provide after data/props
  callHook(vm, 'created')
  // ...
}
```

可以看到 `beforeCreate` 和 `created` 的钩子调用是在 `initState` 的前后，initState 的作用是初始化 props、data、methods、watch、computed 等属性，之后我们会详细分析。那么显然 beforeCreate 的钩子函数中就不能获取到 props、data 中定义的值，也不能调用 methods 中定义的函数。

在这俩个钩子函数执行的时候，并没有渲染 DOM，所以我们也不能够访问 DOM，一般来说，如果组件在加载的时候需要和后端有交互，放在这俩个钩子函数执行都可以，如果是需要访问 props、data 等数据的话，就需要使用 created 钩子函数。

3.4.2. beforeMount & mounted

`beforeMount` 钩子函数发生在 `mount`，也就是 DOM 挂载之前，它的调用时机是在 `mountComponent` 函数中，定义在 `src/core/instance/lifecycle.js` 中：

```
export function mountComponent (
  vm: Component,
  el: ?Element,
  hydrating?: boolean
): Component {
  vm.$el = el
  // ...
  callHook(vm, 'beforeMount')

  let updateComponent
  /* istanbul ignore if */
  if (process.env.NODE_ENV !== 'production' && config.performance && mark) {
    updateComponent = () => {
      const name = vm._name
      const id = vm._uid
      const startTag = `vue-perf-start:${id}`
      const endTag = `vue-perf-end:${id}`

      mark(startTag)
      const vnode = vm._render()
      mark(endTag)
      measure(`vue ${name} render`, startTag, endTag)

      mark(startTag)
```

```
      vm._update(vnode, hydrating)
      mark(endTag)
      measure(`vue ${name} patch`, startTag, endTag)
    }
  } else {
    updateComponent = () => {
      vm._update(vm._render(), hydrating)
    }
  }

  // we set this to vm._watcher inside the watcher's constructor
  // since the watcher's initial patch may call $forceUpdate (e.g. inside child
  // component's mounted hook), which relies on vm._watcher being already defined
  new Watcher(vm, updateComponent, noop, {
    before () {
      if (vm._isMounted) {
        callHook(vm, 'beforeUpdate')
      }
    }
  }, true /* isRenderWatcher */)
  hydrating = false

  // manually mounted instance, call mounted on self
  // mounted is called for render-created child components in its inserted hook
  if (vm.$vnode == null) {
    vm._isMounted = true
    callHook(vm, 'mounted')
  }
  return vm
}
```

在执行 `vm._render()` 函数渲染 VNode 之前，执行了 `beforeMount` 钩子函数，在执行完 `vm._update()` 把 VNode patch 到真实 DOM 后，执行 `mounted` 钩子。注意，这里对 mounted 钩子函数执行有一个判断逻辑，vm.$vnode 如果为 null，则表明这不是一次组件的初始化过程，而是我们通过外部 new Vue 初始化过程。那么对于组件，它的 mounted 时机在哪儿呢?

之前我们提到过，组件的 VNode patch 到 DOM 后，会执行 `invokeInsertHook` 函数，把 `insertedVnodeQueue` 里保存的钩子函数依次执行一遍，它的定义在 `src/core/vdom/patch.js` 中：

```
function invokeInsertHook (vnode, queue, initial) {
  // delay insert hooks for component root nodes, invoke them after the
  // element is really inserted
  if (isTrue(initial) && isDef(vnode.parent)) {
    vnode.parent.data.pendingInsert = queue
  } else {
    for (let i = 0; i < queue.length; ++i) {
      queue[i].data.hook.insert(queue[i])
    }
  }
}
```

该函数会执行 insert 这个钩子函数，对于组件而言，insert 钩子函数的定义在 `src/core/vdom/create-component.js` 中的 `componentVNodeHooks` 中：

```
const componentVNodeHooks = {
  // ...
  insert (vnode: MountedComponentVNode) {
    const { context, componentInstance } = vnode
```

```
    if (!componentInstance._isMounted) {
      componentInstance._isMounted = true
      callHook(componentInstance, 'mounted')
    }
    // ...
  },
}
```

我们可以看到，每个子组件都是在这个钩子函数中执行 `mounted` 钩子函数，并且我们之前分析过，`insertedVnodeQueue` 的添加顺序是先子后父，所以对于同步渲染的子组件而言，`mounted` 钩子函数的执行顺序也是先子后父。

3.4.3. beforeUpdate & Updated

`beforeUpdate` 和 `updated` 的钩子函数执行时机都应该是在数据更新的时候，到目前为止，我们还没有分析 Vue 的数据双向绑定、更新相关，后续会详细介绍。

`beforeUpdate` 的执行时机是在渲染 Watcher 的 before 函数中，我们刚才提到过：

```
export function mountComponent (
  vm: Component,
  el: ?Element,
  hydrating?: boolean
): Component {
  // ...

  // we set this to vm._watcher inside the watcher's constructor
  // since the watcher's initial patch may call $forceUpdate (e.g. inside child
  // component's mounted hook), which relies on vm._watcher being already defined
  new Watcher(vm, updateComponent, noop, {
    before () {
      if (vm._isMounted) {
        callHook(vm, 'beforeUpdate')
      }
    }
  }, true /* isRenderWatcher */)
  // ...
}
```

注意这里有个判断，也就是在组件已经 mounted 之后，才会去调用这个钩子函数。

`update` 的执行时机是在 `flushSchedulerQueue` 函数调用的时候，它的定义在 `src/core/observer/scheduler.js` 中：

```
function flushSchedulerQueue () {
  // ...
  // 获取到 updatedQueue
  callUpdatedHooks(updatedQueue)
}

function callUpdatedHooks (queue) {
  let i = queue.length
  while (i--) {
    const watcher = queue[i]
    const vm = watcher.vm
    if (vm._watcher === watcher && vm._isMounted) {
      callHook(vm, 'updated')
```

```
    }
  }
}
```

后续详细讲到具体的响应式更新后再讲解；

3.4.4. beforeDestroy & destroy

`beforeDestroy` 和 `destroyed` 钩子函数的执行时机在组件销毁的阶段，组件的销毁过程之后会详细介绍，最终会调用 `$destroy` 方法，它的定义在 `src/core/instance/lifecycle.js` 中：

```
Vue.prototype.$destroy = function () {
    const vm: Component = this
    if (vm._isBeingDestroyed) {
      return
    }
    callHook(vm, 'beforeDestroy')
    vm._isBeingDestroyed = true
    // remove self from parent
    const parent = vm.$parent
    if (parent && !parent._isBeingDestroyed && !vm.$options.abstract) {
      remove(parent.$children, vm)
    }
    // teardown watchers
    if (vm._watcher) {
      vm._watcher.teardown()
    }
    let i = vm._watchers.length
    while (i--) {
      vm._watchers[i].teardown()
    }
    // remove reference from data ob
    // frozen object may not have observer.
    if (vm._data.__ob__) {
      vm._data.__ob__.vmCount--
    }
    // call the last hook...
    vm._isDestroyed = true
    // invoke destroy hooks on current rendered tree
    vm.__patch__(vm._vnode, null)
    // fire destroyed hook
    callHook(vm, 'destroyed')
    // turn off all instance listeners.
    vm.$off()
    // remove __vue__ reference
    if (vm.$el) {
      vm.$el.__vue__ = null
    }
    // release circular reference (#6759)
    if (vm.$vnode) {
      vm.$vnode.parent = null
    }
  }
```

`beforeDestroy` 钩子函数的执行时机是在 `$destroy` 函数执行最开始的地方，接着执行了一系列的销毁动作，包括从 `parent` 的 `$children` 中删掉自身，删除 `watcher`，当前渲染的 VNode 执行销毁钩子函数等，执行完毕后再调用 `destroy` 钩子函数。

在 $destroy 的执行过程中，它又会执行 `vm.__patch__(vm._vnode, null)` 触发它子组件的销毁钩子函数，这样一层层的递归调用，所以 destroy 钩子函数执行顺序是先子后父，和 mounted 过程一样。

3.4.5. activated & deactivated

后续讲到 `Keep-alive` 时详细讲解。

3.5. 组件注册

在 Vue.js 中，除了它内置的组件如 `keep-alive、` `component` 、 `transition` 、 `transition-group` 等，其它用户自定义组件在使用前必须注册。很多同学在开发过程中可能会遇到如下报错信息：

```
'Unknown custom element: <xxx> - did you register the component correctly?
 For recursive components, make sure to provide the "name" option.'
```

一般报这个错的原因都是我们使用了未注册的组件。Vue.js 提供了 2 种组件的注册方式，全局注册和局部注册。接下来我们从源码分析的角度来分析这两种注册方式。

3.5.1. 全局注册

要注册一个全局组件，可以使用 `Vue.component(tagName, options)` 。例如：

```
Vue.component('my-component', {
  // 选项
})
```

`Vue.component` 的定义过程发生在最开始初始化 Vue 的全局函数的时候，代码在 s `rc/core/global-api/assets.js` 中：

```
import { ASSET_TYPES } from 'shared/constants'
import { isPlainObject, validateComponentName } from '../util/index'

export function initAssetRegisters (Vue: GlobalAPI) {
  /**
   * Create asset registration methods.
   */
  ASSET_TYPES.forEach(type => {
    Vue[type] = function (
      id: string,
      definition: Function | Object
    ): Function | Object | void {
      if (!definition) {
        return this.options[type + 's'][id]
      } else {
        /* istanbul ignore if */
        if (process.env.NODE_ENV !== 'production' && type === 'component') {
          validateComponentName(id)
        }
        if (type === 'component' && isPlainObject(definition)) {
          definition.name = definition.name || id
```

```
          definition = this.options._base.extend(definition)
        }
        if (type === 'directive' && typeof definition === 'function') {
          definition = { bind: definition, update: definition }
        }
        this.options[type + 's'][id] = definition
        return definition
      }
    }
  })
}
```

函数首先遍历 `ASSET_TYPES`，得到 type 后挂载到 Vue 上 。ASSET_TYPES 的定义在 `src/shared/constants.js` 中：

```
export const ASSET_TYPES = [
  'component',
  'directive',
  'filter'
]
```

所以实际上 Vue 是初始化了 3 个全局函数，并且如果 type 是 `component` 且 definition 是一个对象的话，通过 `this.opitons._base.extend`， 相当于 `Vue.extend` 把这个对象转换成一个继承于 Vue 的构造函数，最后通过 `this.options[type + 's'][id] = definition` 把它挂载到 `Vue.options.components` 上。

由于我们每个组件的创建都是通过 `Vue.extend` 继承而来，我们之前分析过在继承的过程中有这么一段逻辑：

```
Sub.options = mergeOptions(
  Super.options,
  extendOptions
)
```

也就是说它会把 Vue.options 合并到 Sub.options，也就是组件的 options 上， 然后在组件的实例化阶段，会执行 `merge options` 逻辑，把 `Sub.options.components` 合并到 `vm.$options.components` 上。

然后在创建 vnode 的过程中，会执行 _createElement 方法，我们再来回顾一下这部分的逻辑，它的定义在 `src/core/vdom/create-element.js` 中：

```
export function _createElement (
  context: Component,
  tag?: string | Class<Component> | Function | Object,
  data?: VNodeData,
  children?: any,
  normalizationType?: number
): VNode | Array<VNode> {
  // ...
  let vnode, ns
  if (typeof tag === 'string') {
    let Ctor
    ns = (context.$vnode && context.$vnode.ns) || config.getTagNamespace(tag)
    if (config.isReservedTag(tag)) {
      // platform built-in elements
      vnode = new VNode(
        config.parsePlatformTagName(tag), data, children,
        undefined, undefined, context
      )
```

```
    } else if (isDef(Ctor = resolveAsset(context.$options, 'components', tag))) {
      // component
      vnode = createComponent(Ctor, data, context, children, tag)
    } else {
      // unknown or unlisted namespaced elements
      // check at runtime because it may get assigned a namespace when its
      // parent normalizes children
      vnode = new VNode(
        tag, data, children,
        undefined, undefined, context
      )
    }
  } else {
    // direct component options / constructor
    vnode = createComponent(tag, data, context, children)
  }
  // ...
}
```

这里有一个判断逻辑 `isDef(Ctor = resolveAsset(context.$options, 'components', tag))`，先来看一下 `resolveAsset` 的定义，在 `src/core/utils/options.js` 中：

```
/**
 * Resolve an asset.
 * This function is used because child instances need access
 * to assets defined in its ancestor chain.
 */
export function resolveAsset (
  options: Object,
  type: string,
  id: string,
  warnMissing?: boolean
): any {
  /* istanbul ignore if */
  if (typeof id !== 'string') {
    return
  }
  const assets = options[type]
  // check local registration variations first
  if (hasOwn(assets, id)) return assets[id]
  const camelizedId = camelize(id)
  if (hasOwn(assets, camelizedId)) return assets[camelizedId]
  const PascalCaseId = capitalize(camelizedId)
  if (hasOwn(assets, PascalCaseId)) return assets[PascalCaseId]
  // fallback to prototype chain
  const res = assets[id] || assets[camelizedId] || assets[PascalCaseId]
  if (process.env.NODE_ENV !== 'production' && warnMissing && !res) {
    warn(
      'Failed to resolve ' + type.slice(0, -1) + ': ' + id,
      options
    )
  }
  return res
}
```

先通过 `const assets = options[type]` 拿到 assets，然后再尝试拿 assets[id]，这里有个顺序，先直接使用 id 拿，如果不存在，则把 id 变成驼峰的形式再拿，如果仍然不存在则在驼峰的基础上把首字母再变成大写的形式再拿，如果仍然拿

不到则报错。这样说明了我们在使用 `Vue.component(id, definition)` 全局注册组件的时候，id 可以是连字符、驼峰或首字母大写的形式。

那么回到我们的调用 `resolveAsset(context.$options, 'components', tag)`，即拿 `vm.$options.components[tag]`，这样我们就可以在 resolveAsset 的时候拿到这个组件的构造函数，并作为 `createComponent` 的钩子的参数。

3.5.2. 局部注册

Vue.js 也同样支持局部注册，我们可以在一个组件内部使用 components 选项做组件的局部注册，例如

```
import HelloWorld from './components/HelloWorld'

export default {
  components: {
    HelloWorld
  }
}
```

在组件的 Vue 的实例化阶段有一个合并 `option` 的逻辑，之前我们也分析过，所以就把 `components` 合并到 `vm.$options.components` 上，这样我们就可以在 `resolveAsset` 的时候拿到这个组件的构造函数，并作为 `createComponent` 的钩子的参数。

注意，局部注册和全局注册不同的是，只有该类型的组件才可以访问局部注册的子组件，而全局注册是扩展到 `Vue.options` 下，所以在所有组件创建的过程中，都会从全局的 `Vue.options.components` 扩展到当前组件的 `vm.$options.components` 下，这就是全局注册的组件能被任意使用的原因。

3.6. 异步组件

在我们平时的开发工作中，为了减少首屏代码体积，往往会把一些非首屏的组件设计成异步组件，按需加载。Vue 也原生支持了异步组件的能力，如下：

```
Vue.component('async-example', function (resolve, reject) {
  // 这个特殊的 require 语法告诉 webpack
  // 自动将编译后的代码分割成不同的块，
  // 这些块将通过 Ajax 请求自动下载。
  require(['./my-async-component'], resolve)
})
```

我们可以看到，Vue 注册的组件不再是一个对象，而是一个工厂函数，函数有两个参数 resolve 和 reject，函数内部用 setTimout 模拟了异步，实际使用可能是通过动态请求异步组件的 JS 地址，最终通过执行 resolve 方法，它的参数就是我们的异步组件对象。

上一节我们分析了组件的注册逻辑，由于组件的定义并不是一个普通对象，所以不会执行 Vue.extend 的逻辑把它变成一个组件的构造函数，但是它仍然可以执行到 createComponent 函数，我们再来对这个函数做回顾，它的定义在 `src/core/vdom/create-component/js` 中：

```
export function createComponent (
  Ctor: Class<Component> | Function | Object | void,
  data: ?VNodeData,
  context: Component,
  children: ?Array<VNode>,
  tag?: string
): VNode | Array<VNode> | void {
  if (isUndef(Ctor)) {
```

```
    return
  }

  const baseCtor = context.$options._base

  // plain options object: turn it into a constructor
  if (isObject(Ctor)) {
    Ctor = baseCtor.extend(Ctor)
  }

  // ...

  // async component
  let asyncFactory
  if (isUndef(Ctor.cid)) {
    asyncFactory = Ctor
    Ctor = resolveAsyncComponent(asyncFactory, baseCtor, context)
    if (Ctor === undefined) {
      // return a placeholder node for async component, which is rendered
      // as a comment node but preserves all the raw information for the node.
      // the information will be used for async server-rendering and hydration.
      return createAsyncPlaceholder(
        asyncFactory,
        data,
        context,
        children,
        tag
      )
    }
  }
}
```

由于我们这个时候传入的 Ctor 是一个函数，那么它也并不会执行 Vue.extend 逻辑，因此它的 cid 是 undefiend，进入了异步组件创建的逻辑。这里首先执行了 Ctor = resolveAsyncComponent(asyncFactory, baseCtor, context) 方法，它的定义在 `src/core/vdom/helpers/resolve-async-component.js` 中：

```
export function resolveAsyncComponent (
  factory: Function,
  baseCtor: Class<Component>,
  context: Component
): Class<Component> | void {
  if (isTrue(factory.error) && isDef(factory.errorComp)) {
    return factory.errorComp
  }

  if (isDef(factory.resolved)) {
    return factory.resolved
  }

  if (isTrue(factory.loading) && isDef(factory.loadingComp)) {
    return factory.loadingComp
  }

  if (isDef(factory.contexts)) {
    // already pending
    factory.contexts.push(context)
  } else {
```

```
const contexts = factory.contexts = [context]
let sync = true

const forceRender = () => {
  for (let i = 0, l = contexts.length; i < l; i++) {
    contexts[i].$forceUpdate()
  }
}

const resolve = once((res: Object | Class<Component>) => {
  // cache resolved
  factory.resolved = ensureCtor(res, baseCtor)
  // invoke callbacks only if this is not a synchronous resolve
  // (async resolves are shimmed as synchronous during SSR)
  if (!sync) {
    forceRender()
  }
})

const reject = once(reason => {
  process.env.NODE_ENV !== 'production' && warn(
    `Failed to resolve async component: ${String(factory)}` +
    (reason ? `\nReason: ${reason}` : '')
  )
  if (isDef(factory.errorComp)) {
    factory.error = true
    forceRender()
  }
})

const res = factory(resolve, reject)

if (isObject(res)) {
  if (typeof res.then === 'function') {
    // () => Promise
    if (isUndef(factory.resolved)) {
      res.then(resolve, reject)
    }
  } else if (isDef(res.component) && typeof res.component.then === 'function') {
    res.component.then(resolve, reject)

    if (isDef(res.error)) {
      factory.errorComp = ensureCtor(res.error, baseCtor)
    }

    if (isDef(res.loading)) {
      factory.loadingComp = ensureCtor(res.loading, baseCtor)
      if (res.delay === 0) {
        factory.loading = true
      } else {
        setTimeout(() => {
          if (isUndef(factory.resolved) && isUndef(factory.error)) {
            factory.loading = true
            forceRender()
          }
        }, res.delay || 200)
      }
    }
```

```
        if (isDef(res.timeout)) {
          setTimeout(() => {
            if (isUndef(factory.resolved)) {
              reject(
                process.env.NODE_ENV !== 'production'
                  ? `timeout (${res.timeout}ms)`
                  : null
              )
            }
          }, res.timeout)
        }
      }
    }

    sync = false
    // return in case resolved synchronously
    return factory.loading
      ? factory.loadingComp
      : factory.resolved
  }
}
```

这里面核心处理了三种异步组件的创建方法，除了上述例子，还有两种

```
Vue.component(
  'async-webpack-example',
  // 该 `import` 函数返回一个 `Promise` 对象。
  () => import('./my-async-component')
)

const AsyncComp = () => ({
  // 需要加载的组件。应当是一个 Promise
  component: import('./MyComp.vue'),
  // 加载中应当渲染的组件
  loading: LoadingComp,
  // 出错时渲染的组件
  error: ErrorComp,
  // 渲染加载中组件前的等待时间。默认：200ms。
  delay: 200,
  // 最长等待时间。超出此时间则渲染错误组件。默认：Infinity
  timeout: 3000
})
Vue.component('async-example', AsyncComp)
```

接下来依次分析。

3.6.1. 普通的异步组件

针对普通函数的情况，前面几个 if 判断可以忽略，它们是为高级组件所用，进入实际加载逻辑，定义了 `forceRender` 、 `resolve` 和 `reject` 函数，注意 `resolve` 和 `reject` 函数用 `once` 函数做了一层包装，它的定义在 `src/shared/util.js` 中：

```
/**
 * Ensure a function is called only once.
 */
```

```
export function once (fn: Function): Function {
  let called = false
  return function () {
    if (!called) {
      called = true
      fn.apply(this, arguments)
    }
  }
}
```

once 逻辑非常简单，传入一个函数，并返回一个新函数，它非常巧妙地利用闭包和一个标志位保证了它包装的函数只会执行一次，也就是确保 `resolve` 和 `reject` 函数只执行一次。

接下来执行 `const res = factory(resolve, reject)` 逻辑，这块儿就是执行我们组件的工厂函数，同时把 `resolve` 和 `reject` 函数作为参数传入，组件的工厂函数通常会先发送请求去加载我们的异步组件的 JS 文件，拿到组件定义的对象 res 后，执行 resolve(res) 逻辑，它会先执行 `factory.resolved = ensureCtor(res, baseCtor)`

```
function ensureCtor (comp: any, base) {
  if (
    comp.__esModule ||
    (hasSymbol && comp[Symbol.toStringTag] === 'Module')
  ) {
    comp = comp.default
  }
  return isObject(comp)
    ? base.extend(comp)
    : comp
}
```

这个函数目的是为了保证能找到异步组件 JS 定义的组件对象，并且如果它是一个普通对象，则调用 `Vue.extend` 把它转换成一个组件的构造函数。

`resolve` 逻辑最后判断了 sync，显然我们这个场景下 sync 为 false，那么就会执行 forceRender 函数，它会遍历 factory.contexts，拿到每一个调用异步组件的实例 vm, 执行 `vm.$forceUpdate()` 方法，它的定义在 `src/core/instance/lifecycle.js` 中：

```
  Vue.prototype.$forceUpdate = function () {
    const vm: Component = this
    if (vm._watcher) {
      vm._watcher.update()
    }
  }
```

`$forceUpdate` 调用渲染 watcher 的 update 方法，让渲染 watcher 对应的回调函数执行，也就是触发了组件的重新渲染。之所以这么做是因为 Vue 通常是数据驱动视图重新渲染，但是在整个异步组件加载过程中是没有数据发生变化的，所以通过执行 $forceUpdate 可以强制组件重新渲染一次。

3.6.2. Promise异步组件

```
Vue.component(
  'async-webpack-example',
  // 该 `import` 函数返回一个 `Promise` 对象。
  () => import('./my-async-component')
)
```

webpack 2+ 支持了异步加载的语法糖：`() => import('./my-async-component')`，当执行完 `res = factory(resolve, reject)`，返回的值就是 import('./my-async-component') 的返回值，它是一个 Promise 对象。接着进入 if 条件，又判断了 `typeof res.then === 'function'`，条件满足，执行：

```
if (isUndef(factory.resolved)) {
  res.then(resolve, reject)
}
```

当组件异步加载成功后，执行 `resolve`，加载失败则执行 `reject`，这样就非常巧妙地实现了配合 webpack 2+ 的异步加载组件的方式（Promise）加载异步组件。

3.6.3. 高级异步组件

由于异步加载组件需要动态加载 JS，有一定网络延时，而且有加载失败的情况，所以通常我们在开发异步组件相关逻辑的时候需要设计 loading 组件和 error 组件，并在适当的时机渲染它们。Vue.js 2.3+ 支持了一种高级异步组件的方式，它通过一个简单的对象配置，帮你搞定 loading 组件和 error 组件的渲染时机，你完全不用关心细节，非常方便。接下来我们就从源码的角度来分析高级异步组件是怎么实现的。

```
const AsyncComp = () => ({
  // 需要加载的组件。应当是一个 Promise
  component: import('./MyComp.vue'),
  // 加载中应当渲染的组件
  loading: LoadingComp,
  // 出错时渲染的组件
  error: ErrorComp,
  // 渲染加载中组件前的等待时间。默认：200ms。
  delay: 200,
  // 最长等待时间。超出此时间则渲染错误组件。默认：Infinity
  timeout: 3000
})
Vue.component('async-example', AsyncComp)
```

高级异步组件的初始化逻辑和普通异步组件一样，也是执行 `resolveAsyncComponent`，当执行完 `res = factory(resolve, reject)`，返回值就是定义的组件对象，显然满足 else if (isDef(res.component) && typeof res.component.then === 'function') 的逻辑，接着执行 res.component.then(resolve, reject)，当异步组件加载成功后，执行 resolve，失败执行 reject。

因为异步组件加载是一个异步过程，它接着又同步执行了如下逻辑：

```
if (isDef(res.error)) {
  factory.errorComp = ensureCtor(res.error, baseCtor)
}

if (isDef(res.loading)) {
  factory.loadingComp = ensureCtor(res.loading, baseCtor)
  if (res.delay === 0) {
    factory.loading = true
  } else {
    setTimeout(() => {
      if (isUndef(factory.resolved) && isUndef(factory.error)) {
        factory.loading = true
        forceRender()
      }
    }, res.delay || 200)
  }
}
```

```
if (isDef(res.timeout)) {
  setTimeout(() => {
    if (isUndef(factory.resolved)) {
      reject(
        process.env.NODE_ENV !== 'production'
        ? `timeout (${res.timeout}ms)`
        : null
      )
    }
  }, res.timeout)
}
```

先判断 `res.error` 是否定义了 error 组件，如果有的话则赋值给 `factory.errorComp` 。 接着判断 `res.loading` 是否定义了 loading 组件，如果有的话则赋值给 `factory.loadingComp` ，如果设置了 res.delay 且为 0，则设置 factory.loading = true，否则延时 delay 的时间执行

```
if (isUndef(factory.resolved) && isUndef(factory.error)) {
    factory.loading = true
    forceRender()
}
```

最后判断 res.timeout，如果配置了该项，则在 res.timout 时间后，如果组件没有成功加载，执行 reject。

在 resolveAsyncComponent 的最后有一段逻辑：

```
sync = false
return factory.loading
  ? factory.loadingComp
  : factory.resolved
```

如果 delay 配置为 0，则这次直接渲染 loading 组件，否则则延时 delay 执行 forceRender，那么又会再一次执行到 `resolveAsyncComponent` 。

那么这时候我们有几种情况，按逻辑的执行顺序，对不同的情况做判断：

3.6.3.1. 异步组件加载失败

当异步组件加载失败，会执行 `reject` 函数：

```
const reject = once(reason => {
  process.env.NODE_ENV !== 'production' && warn(
    `Failed to resolve async component: ${String(factory)}` +
    (reason ? `\nReason: ${reason}` : '')
  )
  if (isDef(factory.errorComp)) {
    factory.error = true
    forceRender()
  }
})
```

这个时候会把 factory.error 设置为 true，同时执行 `forceRender()` 再次执行到 `resolveAsyncComponent` ：

```
if (isTrue(factory.error) && isDef(factory.errorComp)) {
  return factory.errorComp
}
```

那么这个时候就返回 `factory.errorComp`，直接渲染 `error` 组件。

3.6.3.2. 异步组件加载成功

当异步组件加载成功，会执行 `resolve` 函数：

```
const resolve = once((res: Object | Class<Component>) => {
  factory.resolved = ensureCtor(res, baseCtor)
  if (!sync) {
    forceRender()
  }
})
```

首先把加载结果缓存到 `factory.resolved` 中，这个时候因为 sync 已经为 false，则执行 `forceRender()` 再次执行到 `resolveAsyncComponent`：

```
if (isDef(factory.resolved)) {
  return factory.resolved
}
```

那么这个时候直接返回 `factory.resolved`，渲染成功加载的组件。

3.6.3.3. 异步组件加载中

如果异步组件加载中并未返回，这时候会走到这个逻辑：

```
if (isTrue(factory.loading) && isDef(factory.loadingComp)) {
  return factory.loadingComp
}
```

那么则会返回 `factory.loadingComp`，渲染 `loading` 组件。

3.6.3.4. 异步组件加载超时

如果超时，则走到了 `reject` 逻辑，之后逻辑和加载失败一样，渲染 error 组件。

3.6.4. 异步组件 patch

回到 `createComponent` 的逻辑：

```
Ctor = resolveAsyncComponent(asyncFactory, baseCtor, context)
if (Ctor === undefined) {
  return createAsyncPlaceholder(
    asyncFactory,
    data,
    context,
    children,
    tag
  )
}
```

如果是第一次执行 `resolveAsyncComponent`，除非使用高级异步组件 `0 delay` 去创建了一个 `loading` 组件，否则返回是 `undefiend`，接着通过 `createAsyncPlaceholder` 创建一个注释节点作为占位符。它的定义在 `src/core/vdom/helpers/resolve-async-components.js` 中：

```
export function createAsyncPlaceholder (
  factory: Function,
  data: ?VNodeData,
  context: Component,
  children: ?Array<VNode>,
  tag: ?string
): VNode {
  const node = createEmptyVNode()
  node.asyncFactory = factory
  node.asyncMeta = { data, context, children, tag }
  return node
}
```

实际上就是就是创建了一个占位的注释 `VNode`，同时把 `asyncFactory` 和 `asyncMeta` 赋值给当前 `vnode`。

当执行 `forceRender` 的时候，会触发组件的重新渲染，那么会再一次执行 `resolveAsyncComponent`，这时候就会根据不同的情况，可能返回 loading、error 或成功加载的异步组件，返回值不为 undefined，因此就走正常的组件 render、patch 过程。

3.6.5. 总结

通过上面部分，我们知道了 3 种异步组件的实现方式，它实现了 `loading`、`resolve`、`reject`、`timeout` 4 种状态。异步组件实现的本质是 2 次渲染，除了 `0 delay` 的高级异步组件第一次直接渲染成 `loading` 组件外，其它都是第一次渲染生成一个注释节点，当异步获取组件成功后，再通过 `forceRender` 强制重新渲染，这样就能正确渲染出我们异步加载的组件了。