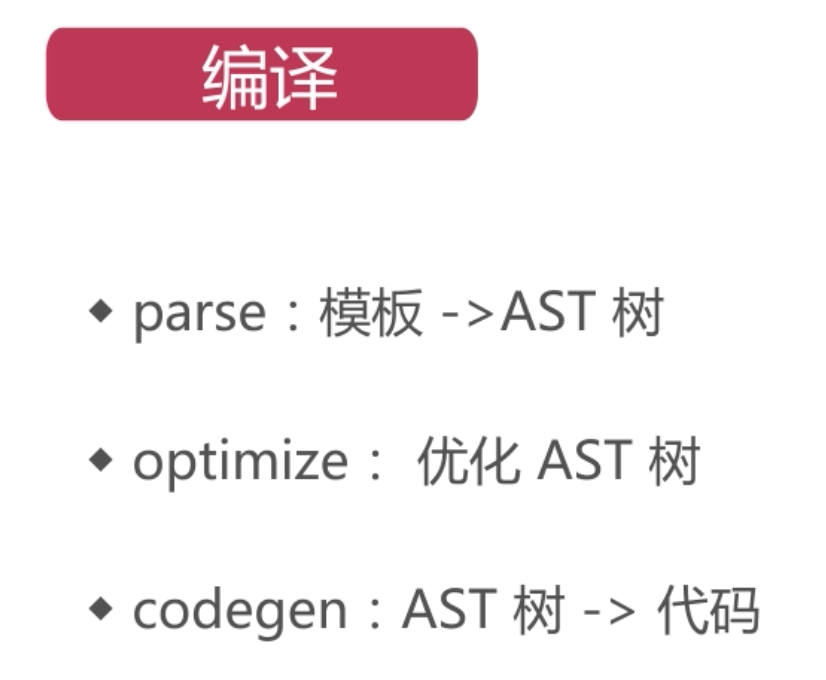
VUE源码解析整理

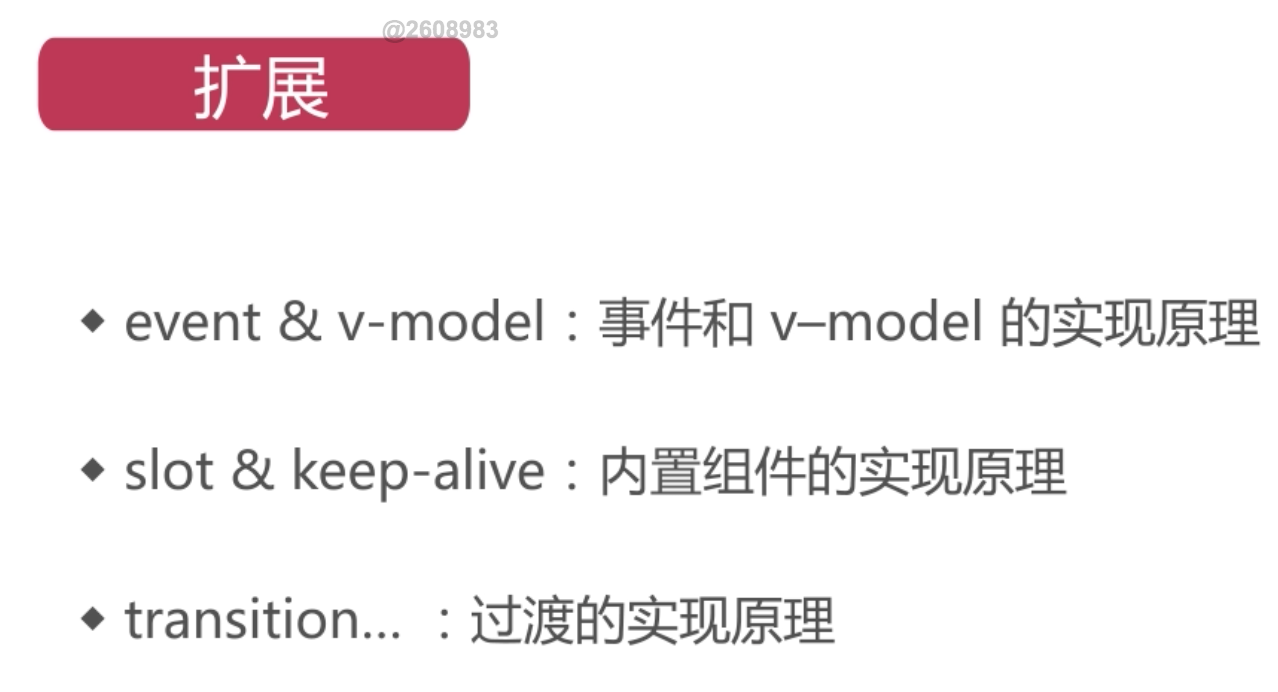
# 准备工作

* 课程内容：











* flow

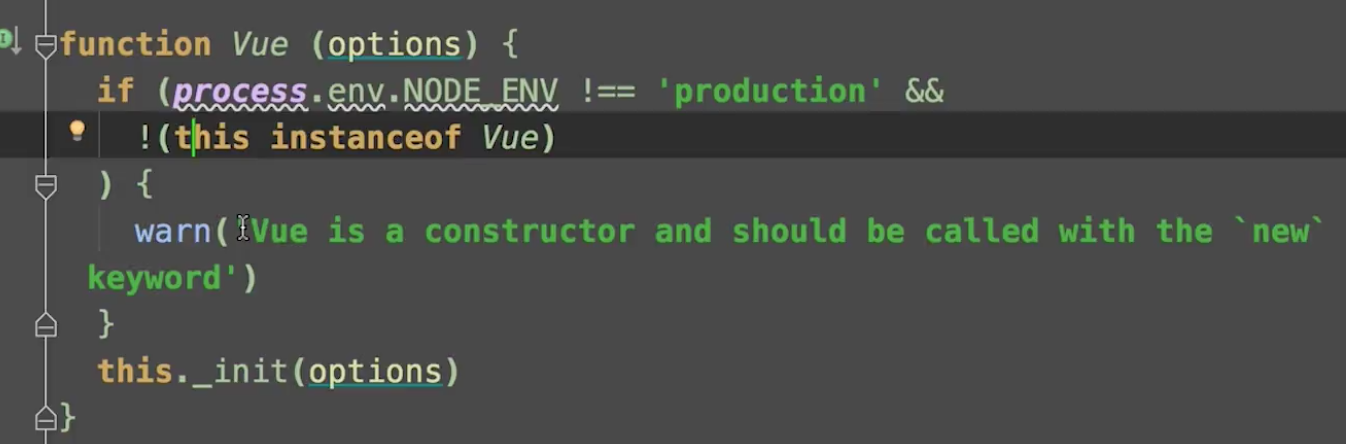
全局安装flow

npm install flow-bin –g

在当前目录下创建.flowconfig文件

命令：flow init

* Vue入口



本质上就是一个用 Function 实现的 Class，然后它的原型 prototype 以及它本身都扩展了一系列的方法和属性。

# 数据驱动

* 数据驱动

Vue.js 一个核心思想是数据驱动。所谓数据驱动，是指视图是由数据驱动生成的，我们对视图的修改，不会直接操作DOM，而是通过修改数据。它相比我们传统的前端开发，如使用jQuery等前端库直接修改DOM，大大简化了代码量。特别是当交互复杂的时候，只关心数据的修改会让代码的逻辑变的非常清晰，因为DOM变成了数据的映射，我们所有的逻辑都是对数据的修改，而不用碰触DOM，这样的代码非常利于维护。

* new Vue发生了什么

Vue 初始化主要就干了几件事情，合并配置，初始化生命周期，初始化事件中心，初始化渲染，初始化 data、props、computed、watcher 等等。

Vue的初始化逻辑写的非常清楚，把不同的功能逻辑拆成一些单独的函数执行，让主线逻辑一目了然，这样的编程思想是非常值得借鉴和学习的。

由于我们这一章的目标是弄清楚模板和数据如何渲染成最终的DOM，所以各种初始化逻辑我们先不看。在初始化的最后，检测到如果有el属性，则调用vm.$mount方法挂载vm，挂载的目标就是把模板渲染成最终的DOM，那么接下来我们来分析 Vue的挂载过程。

* Vue实例挂载

mountComponent方法的逻辑也是非常清晰的，它会完成整个渲染工作，接下来我们要重点分析其中的细节，也就是最核心的2个方法：vm.\_render和vm.\_update。

* Render

vm.\_render最终是通过执行createElement方法并返回的是vnode，它是一个虚拟Node。

* Virtual DOM

其实 VNode是对真实DOM的一种抽象描述，它的核心定义无非就几个关键属性，标签名、数据、子节点、键值等，其它属性都是都是用来扩展VNode的灵活性以及实现一些特殊feature的。由于VNode只是用来映射到真实DOM的渲染，不需要包含操作DOM的方法，因此它是非常轻量和简单的。

Virtual DOM除了它的数据结构的定义，映射到真实的DOM实际上要经历 VNode 的 create、diff、patch等过程。那么在Vue.js中，VNode的create是通过之前提到的createElement方法创建的，我们接下来分析这部分的实现。

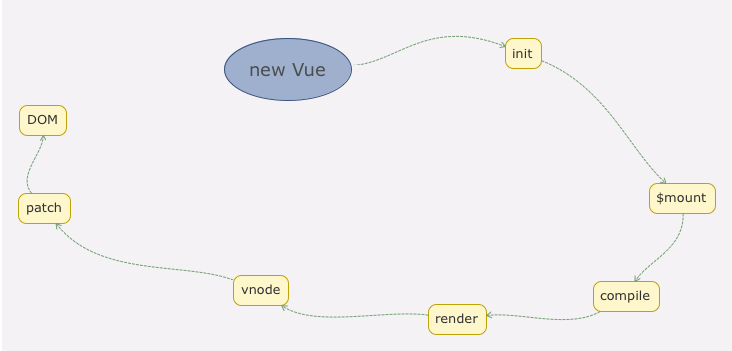
* createElement

createElement创建VNode的过程，每个VNode有children，children每个元素也是一个VNode，这样就形成了一个VNode Tree，它很好的描述了我们的 DOM Tree。

回到mountComponent函数的过程，我们已经知道vm.\_render是如何创建了一个VNode，接下来就是要把这个VNode渲染成一个真实的DOM并渲染出来，这个过程是通过vm.\_update完成的，接下来分析一下这个过程。

* update

从初始化 Vue 到最终渲染的整个过程。



我们这里只是分析了最简单和最基础的场景，在实际项目中，我们是把页面拆成很多组件的，Vue另一个核心思想就是组件化。

# 组件化

* 组件化

Vue.js 另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。

我们在用Vue.js开发实际项目的时候，就是像搭积木一样，编写一堆组件拼装生成页面。在Vue.js的官网中，也是花了大篇幅来介绍什么是组件，如何编写组件以及组件拥有的属性和特性。

* createComponent

createComponent的实现，它在渲染一个组件的时候的3个关键逻辑：构造子类构造函数，安装组件钩子函数和实例化vnode。createComponent后返回的是组件vnode，它也一样走到vm.\_update方法，进而执行了patch函数。

* patch
* 学习目标
* 了解组件patch的整体流程
* 了解组件patch流程中的activeInstance、vm.$vnode、vm.\_vnode等
* 了解嵌套组件的插入顺序
* 总结
* patch的整体流程：createComponent（返回true）->子组件初始化（init，createComponentForVnode,合并lifecycle）->子组件render->子组件patch(子组件中还有组件，递归调用createComponent)
* activeInstance为当前激活的vm实例（子组件创建过程，作为parent传入，组件创建就是一个深度遍历过程，在遍历过程中很好维护父子关系）；vm.$vnode为组件的占位vnode；vm.vnode为组件的渲染vnode。
* 嵌套组件的插入（插入是DOM的插入）顺序是先子后父
* 合并配置
* 学习目标
* 了解外部调用场景的配置合并
* 了解组件现场场景的配置合并
* 总结
* 外部调用场景下（我们new vue()时候）的合并配置是通过mergeOption,并遵循一定的合并策略；
* 组件合并是通过initInternalComponent,它的合并更快（先把Vue的options和组件定义的options进行合并，再把这个options和我们定义的options进行合并）
* 框架、库的设计都是类似，自身定义了默认配置，同事可以在初始化阶段传入配置，然后merge配置，来达到定制化不同需求的目的。
* 生命周期
* 学习目标
* 了解Vue.js有哪些生命周期
* 了解各个生命周期的执行时机及完们能做的事情
* 总结
* Vue.js的生命周期函数就是在初始化及数据更新过程各个阶段执行不同的钩子函数
* 在created钩子函数中可以访问到数据，在mounted够子函数中可以访问到DOM，在destroyed钩子函数中可以做一些定时器销毁工作。
* 组件注册
* 学习目标
* 了解组件注册的2种方式-全局注册和局部注册
* 了解2种注册方式的差异
* 总结
* 全局注册（在Vue上注册）的组件可以任意地方使用，而局部注册（在sub上注册）的组件只能在当前组件内使用
* 通常组件库中的基础组件建议全局注册，而业务组件建议局部注册
* 异步组件
* 学习目标
* 了解一步组件的实现原理
* 了解一步组件的3种实现方式
* 总结
* 异步组件实现的本质是2次渲染，先渲染成注释节点，当组件加载成功后，再通过forceRender重新渲染。
* 异步组件3中实现方式中，高级异步组件的设计是非常巧妙的，它可以通过简单的配置实现了loading、resolve、reject、timeout4中状态。

# 深入响应式原理（上）

* 深入响应式原理

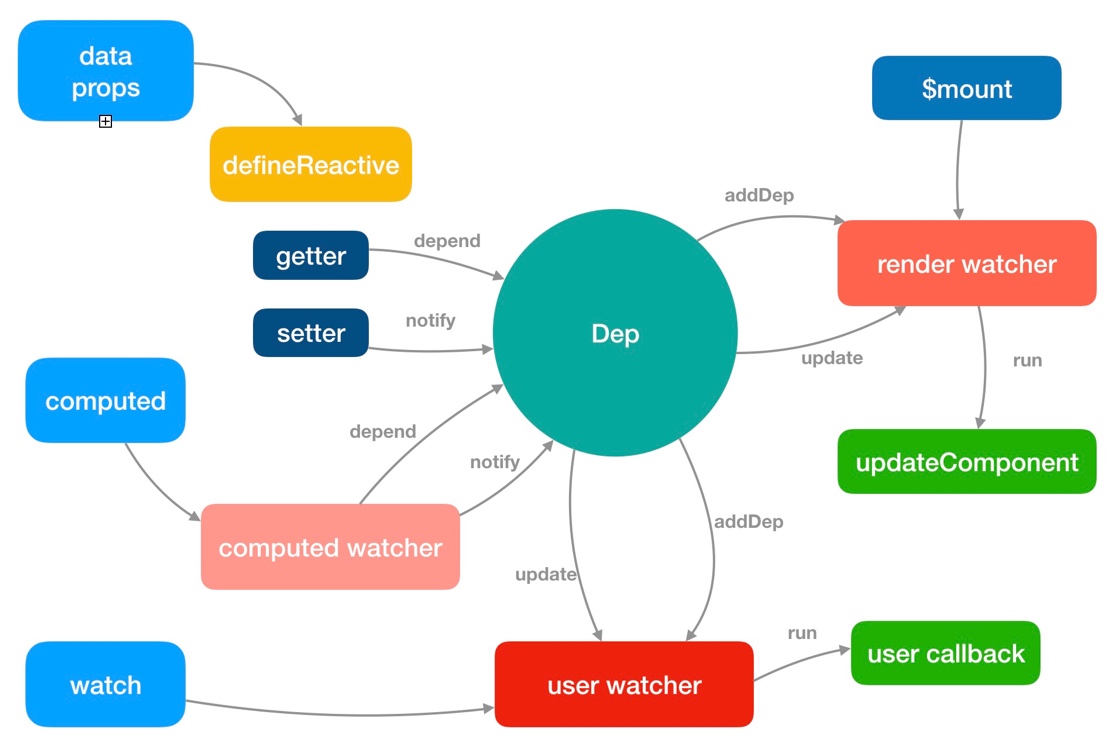
前面 2 章介绍的都是Vue怎么实现数据渲染和组件化的，主要讲的是初始化的过程，把原始的数据最终映射到DOM中，但并没有涉及到数据变化到DOM变化的部分。而Vue的数据驱动除了数据渲染DOM之外，还有一个很重要的体现就是数据的变更会触发DOM的变化。

其实前端开发最重要的2个工作，一个是把数据渲染到页面，另一个是处理用户交互。Vue把数据渲染到页面的能力我们已经通过源码分析出其中的原理了，但是由于一些用户交互或者是其它方面导致数据发生变化重新对页面渲染的原理我们还未分析。

* 响应式对象
* 学习目标
* 了解什么是响应式对象；
* 了解响应式对象的创建过程。
* 总结
* 响应式对象核心是利用了Object.defineProperty给对象的属性添加getter和setter。
* Vue会把props、data等变成响应式对象，在创建过程中，发现子属性也为对象则递归把该对象变成响应式。
* 依赖收集
* 学习目标
* 了解什么是依赖收集
* 了解依赖收集的流程以及它的目的
* 总结
* 依赖收集就是订阅数据变化watcher的收集；
* 依赖收集的目的是为了当这些响应式数据发生变化，触发它们的setter的时候，能知道应该通知哪些订阅者去做相应的逻辑处理。
* 派发更新
* 学习目标
* 了解什么是派发更新
* 了解派发更新的流程以及其中做的一些优化
* 总结
* 派发更新就是当数据发生变化后，通知所有订阅了这个数据变化的watcher执行update。
* 派发更新的过程中会把所有要执行update的watcher推入到队列中（同一个watcher不会重复执行），在nextTick后执行flush(异步只执行一次)。
* nextTick
* 学习目标
* 了解nextTick的实现原理
* 了解数据变化到DOM的变化是异步过程
* 检测变化的注意事项
* 学习目标
* 了解哪些数据的变化不能被检测到
* 了解如何解决数据变化监测的问题以及其中的原理
* 总结
* 响应式数据中对于对象新增删除属性以及数组的下标访问修改和添加数据等的变化观测不到。
* 通过Vue.set以及数组的API可以解决这些问题，本质上它们内部手动去做了依赖更新的派发。

# 深入响应式原理（下）

* 计算属性VS侦听属性
* 学习目标
* 了解计算属性的实现原理
* 了解侦听属性的实现原理和几种配置
* 了解他们各自的适用场景
* 总结
* 计算属性的本质是computed watcher
* 侦听属性的本质是user watcher，它还支持deep、sync、immediate等配置
* 计算属性适合用在模板渲染中，某个值是依赖了其它的响应式对象甚至计算属性计算而来；而侦听属性适用于观测某个值的变化去完成一段复杂的业务逻辑。
* 组件更新
* 学习目标
* 了解组件更新过程
* 了解更新过程中虚拟DOM的diff算法
* 总结
* 组件更新的过程核心就是新旧vnode diff，对新旧节点相同以及不同的情况分别做不同的处理。
* 新旧节点不同的更新流程是创建新节点->更新父占位符节点->删除旧节点
* 新旧节点相同的更新流程是去获取他们的children，根据不同情况做不同的更新逻辑。
* 原理图

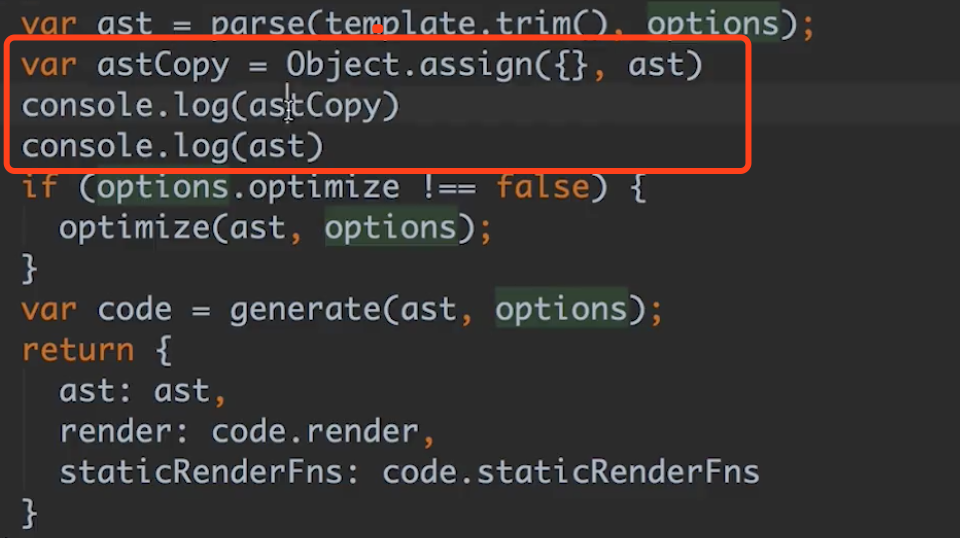


VUE响应式原理就是去观察我们数据的变化，当我们这些数据发生变化以后，能通知到我们对应的观察者，来实现相关的逻辑，整个响应式原理最核心的就是Dep的实现，它实际上就是连接这些数据和他们对应观察者的桥梁，在VUE初始化阶段，他会对VUE上定义的不同的属性做一些处理，对data和props通过observe和defineReactive等一系列的操作，把data和props每个属性都变成响应式，同时内部会持有一个dep的实现。当我们访问到这些数据的时候，就会触发Dep的depend的方法，来收集依赖，收集的依赖是当前计算的watcher，当前计算的watcher也就是Dep target就会作为订阅者，然后订阅这些数据的变化。当我们去修改这些数据的时候，我们就会通知调用Dep的notify，来做update的逻辑。对于computed的逻辑，内部会创建一个computed watcher这样一个watcher,这个watcher会持有dep这样的一个实例，当我们去访问这个computed属性时候，computed的通过evaluate的计算方法，这个时候，他就会触发内部的depend计算方法，去收集依赖，和数据一样，他也会收集当时我们计算的watcher，把当时计算的watcher作为DEP的subscribe（也就是订阅着），收集起来，收集起来作用是当我们计算属性的依赖值发生变化以后，会触发computed watcher重新计算，重新计算如果计算结果变了，他也会调用Dep的notify，通知订阅这个computed的订阅者，他们的更新。对于watcher而言，他实际上就是创建一个user watcher，user watcher可以理解为用户的watcher，用户自己定义的watcher，他可以观察我们data的变化，也可以观察计算属性的变化，我们创建的这个watcher去观察某个watcher或着某个计算属性发生变化，就会通知某个Dep，去遍历所有的user watcher,去调用所有的update方法，求值之后发现新旧值发生变化以后,就会调用run，执行我们用户调用的user callback。数据的渲染，包括它的重新渲染，是基于响应式系统的，因为在整个VUE的创建过程中，对每一个组件而言，都会执行组建的$mount方法，$mount过程内部会创建唯一一个的render watcher，这个watcher当他执行render时候，也就是去渲染Vnode过程中，会访问我们的data或者计算属性，render watcher就做为一个订阅者，订阅了数据和计算属性的变化，一但他们发生变化以后，就会触发setter的notify和computed watcher里面的Dep notify,触发它们以后，就会触发render watcher里面的update,然后执行它们的run方法，然后render watcher执行run方法的时候，最终就会调用updateComponent方法，最后就会重新做一次渲染。

# 编译（上）

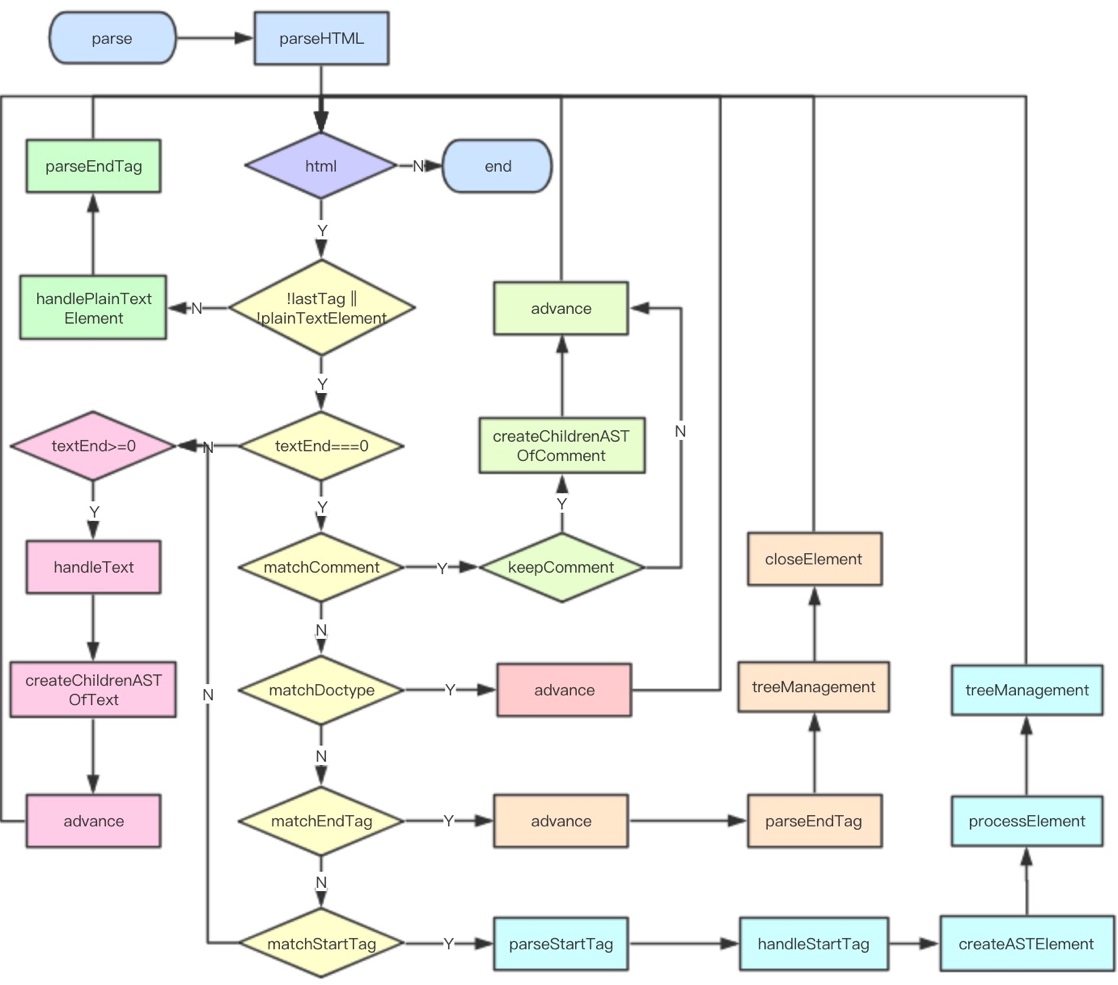
* 编译

之前我们分析过模板到真实DOM渲染的过程，中间有一个环节是把模板编译成render函数，这个过程我们把它称作编译



引用类型的打印的话得新生成的一个对象，否则后面会对数据进行修改

* 编译入口
* 学习目标
* 了解编译入口函数的位置
* 了解Vue.js为什么要这么设计编译入口
* 总结
* Vue.js在不同的平台下都会有编译的过程，因此编译过程中的依赖的配置baseOptions会有所不同
* Vue.js利用了函数柯里化的技巧很好的实现了baseOptions的参数保留。同样，Vue.js也是利用函数柯里化技巧把baseCompile函数抽出来，把真正便利的过程和其它逻辑如对编译配置处理、缓存处理等剥离开。
* parse
* 学习目标
* 了解parse的目标；
* 了解parse的整体流程
* 了解AST的创建过程
* 总结



* 首先会执行parse，实际上就是执行parseHTML，对文本做一个解析的过程，parseHTML就会对传入的HTML做一个判断，如果不为空，就做后面的流程，直到它为空就结束（while循环），开始做如下判断，
* 有没有lastTag(不断保留上次的尾标签，一开始为空)或着plainTextElement(scripth和style标签)，
* textEnd(就是我们查找<的位置),如果文本一开始就是<,就做如下判断：
* 是否matchComment(注释节点)，keepComment,保留注释内容，创建一个注释的AST节点（createChildrenASTOfComment）,执行advance（前进我们的HTML，然后HTML往后截断）
* 是否matchDoctype(Doctype节点),只做advance前进
* 是否matchEndTag(结束标签)，先advance，parseEndTag(对结束标签做解析)，treeManagement(对AST树作管理)，closeElement
* 是否matchStartTag(开始标签)，parseStartTag(对开始标签做一些处理，解析出标签名，属性等)，handleStartTag(对开始标签做一些处理)，createASTElement(创建一个元素AST节点)，processElement（对AST树做一些处理），treeManagement（对树进行管理）
* textEnd大于等于0时候，textEnd>=0,说明<在我们的文本中，handleText（对文本做一些处理），createChildrenASTOfText(创建一个文本的AST节点)，advance（继续前进）
* parse的目标是把template模板解析成AST树
* 整个parse过程就是利用正则表达式顺序解析模板，对于不同的标签和节点处理过程，直至模板解析完毕。
* 在解析过程中，遇到不同的标签和节点会执行对应的处理函数，来达到构建AST树的目的。

# 编译（下）

* optimize
* 学习目标
* 了解optimize目标
* 了解optimize的实现过程
* 总结
* optimize的目标是通过标记静态根的方式，优化重新渲染过程中对静态节点的处理逻辑。
* optimize的过程就是深度遍历这个AST树，先标记静态节点，再标记静态根。
* codegen
* 学习目标
* 了解codegen目标（把AST树转换成可执行的代码字符串）
* 了解codegen的整体流程

# 扩展（上）

* 扩展

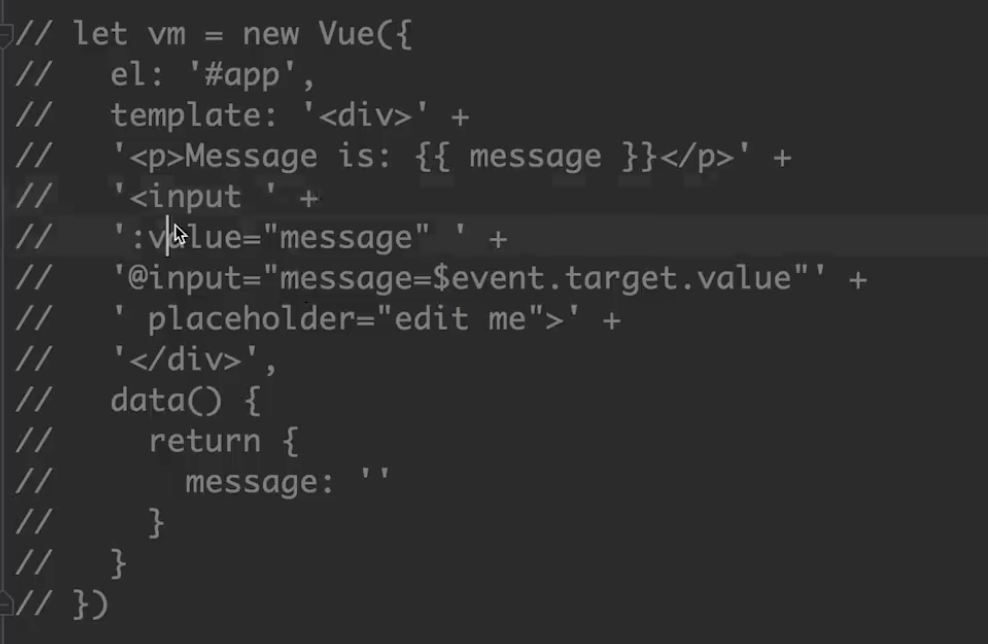
Vue 还提供了很多好用的feature如event、v-model、slot、keep-alive、transition 等等。对他们的理解有助于我们在平时开发中更好地应用这些feature，即使出现bug我们也可以很从容地应对。

* event
* 学习目标
* 了解event的实现原理
* 了解DOM事件和自定义事件的区别
* 总结
* event在编译阶段生成相关的data，对于DOM事件在patch过程中的阶段和更新阶段执行updateDOMListeners生产DOM事件（或者更新DOM事件回调函数）；对于自定义事件，会在组建初始化阶段通过initEvents创建。
* 原生DOM事件和自定义事件主要区别在于添加和删除事件的方式不一样，并且在自定义事件的派发是往当前实例上派发，但是可以利用在父组建环境中定义回调函数来实现父子组建的通讯。

# 扩展（中）

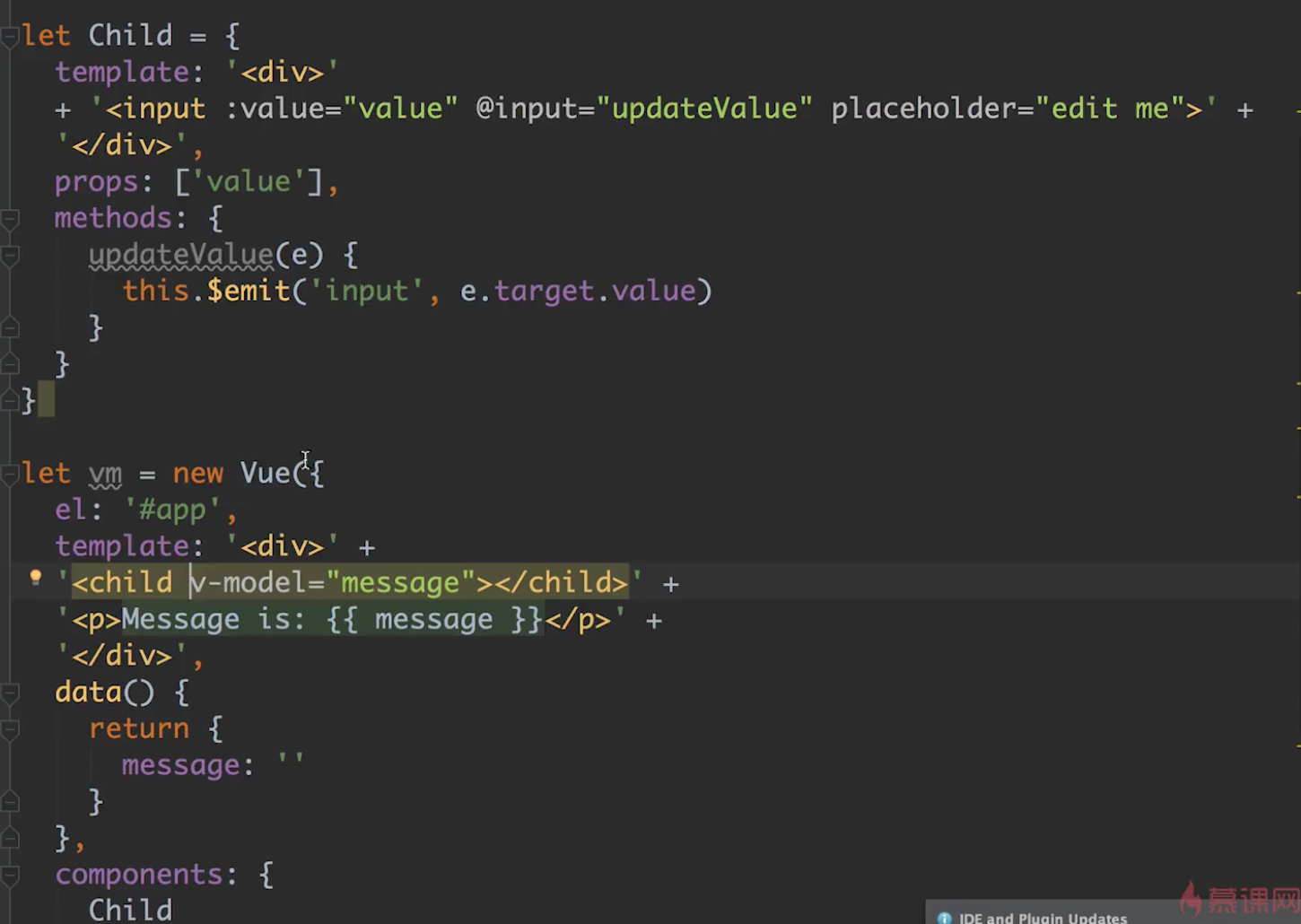
* v-model
* 学习目标
* 了解v-model的本质
* 了解v-model的实现原理
* 总结
* v-model的本质就是语法糖，但是运行时也做了一些优化
* v-model即可以支持原生表单元素，也可以支持组件。在组建的实现中，我们时可以配置子组件接受的prop名称，以及派发的事件名称。

  
约等于下面

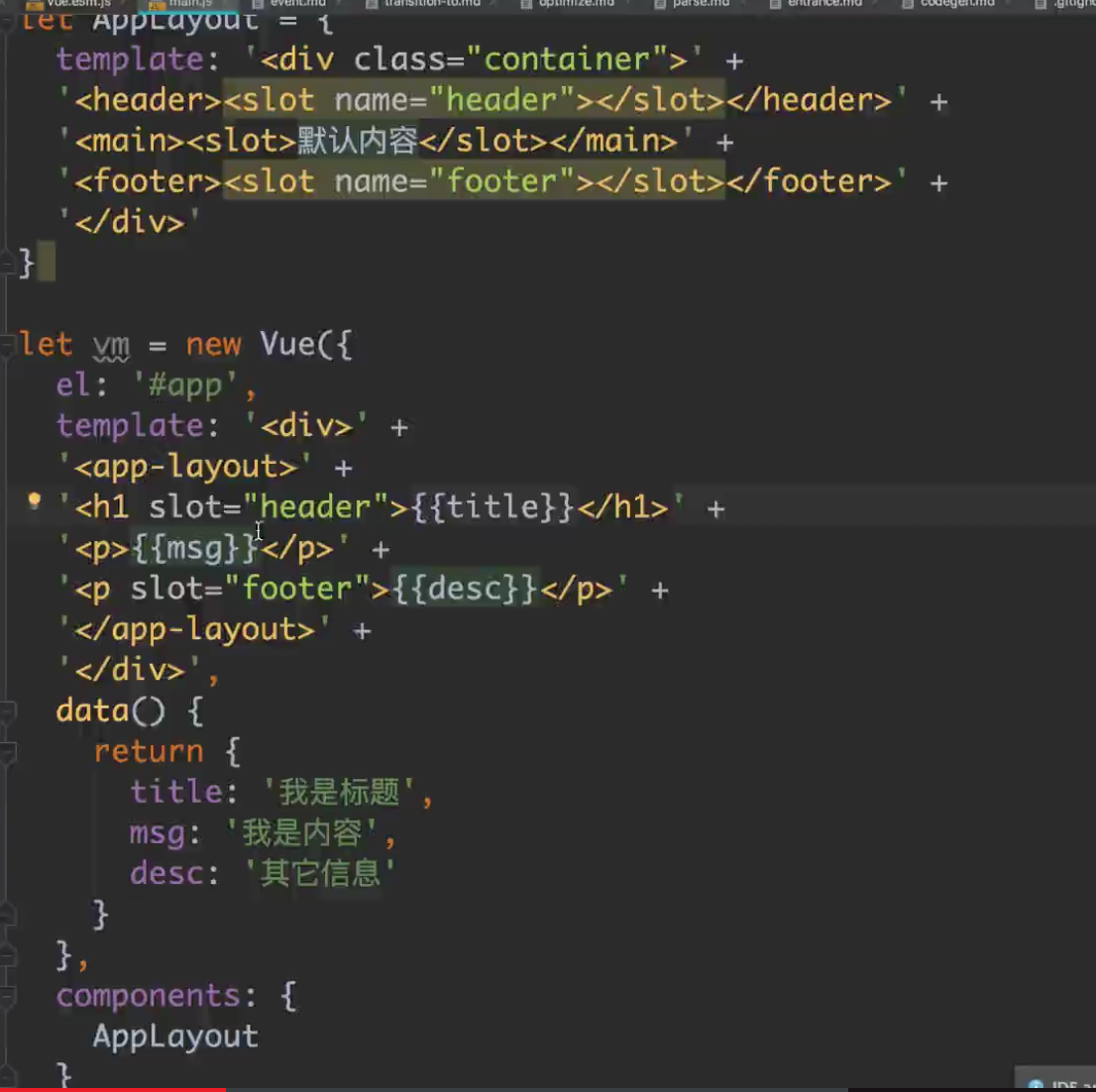


使用v-model之后，会在props上面添加value属性

组件使用v-model

****

* slot
* 学习目标
* 总结
* 普通插槽：在父组件编译和渲染阶段生成vnodes，所以数据的作用域是父组件实例，子组件渲染的时候直接拿到这些渲染好的vnodes。
* 作用域插槽：父组件在编译和渲染阶段并不会直接生成vnodes，而是在父节点vnode的data中保留一个scopedSlots对象，存储着不同名称的插槽以及它们对应的渲染函数，直邮在渲染子组件阶段才会执行这个渲染函数生成vnodes

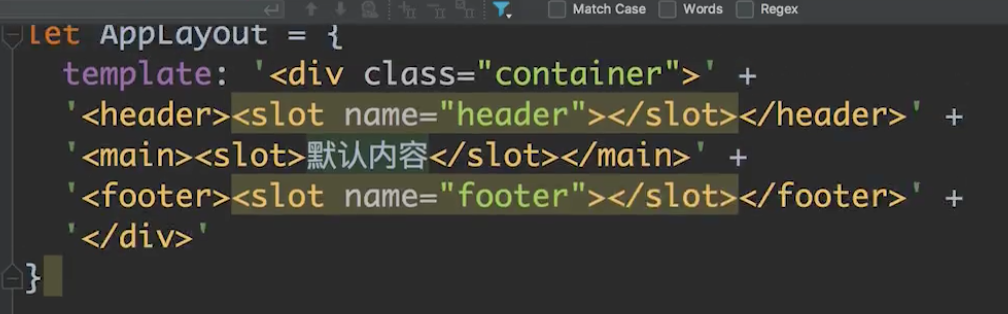
****

header、main、footer原生标签

slot标签里面可以写一些默认内容。

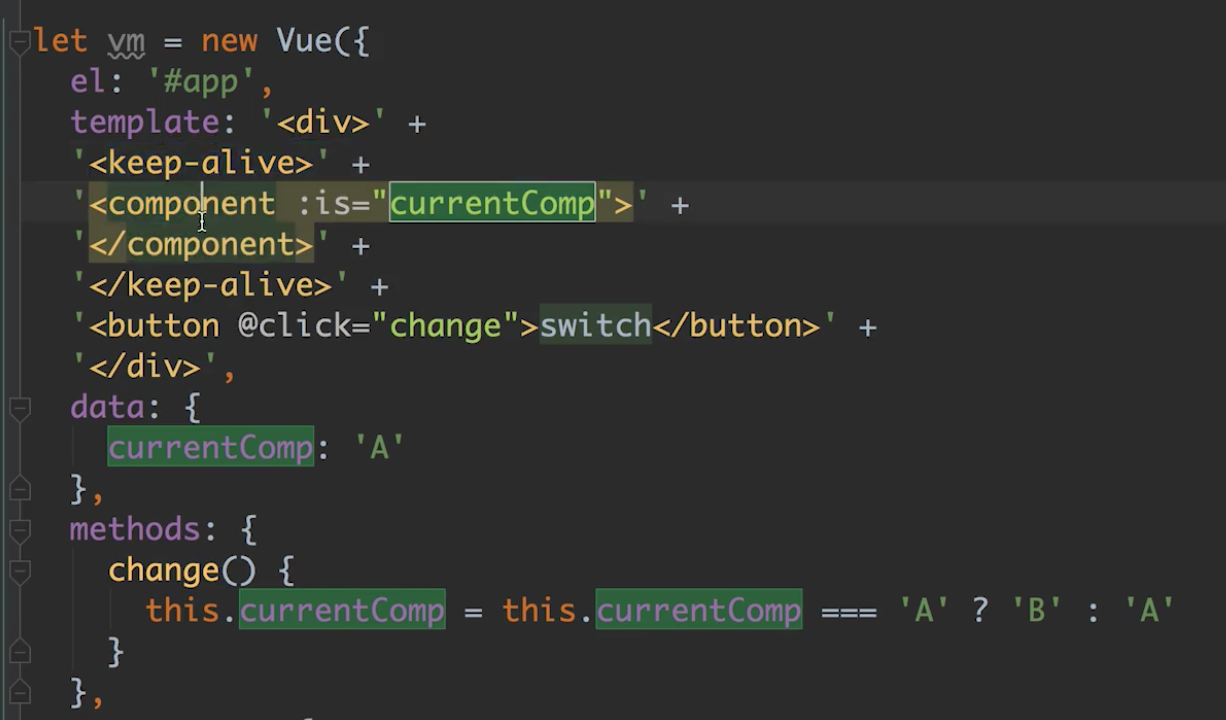
插槽数据更新





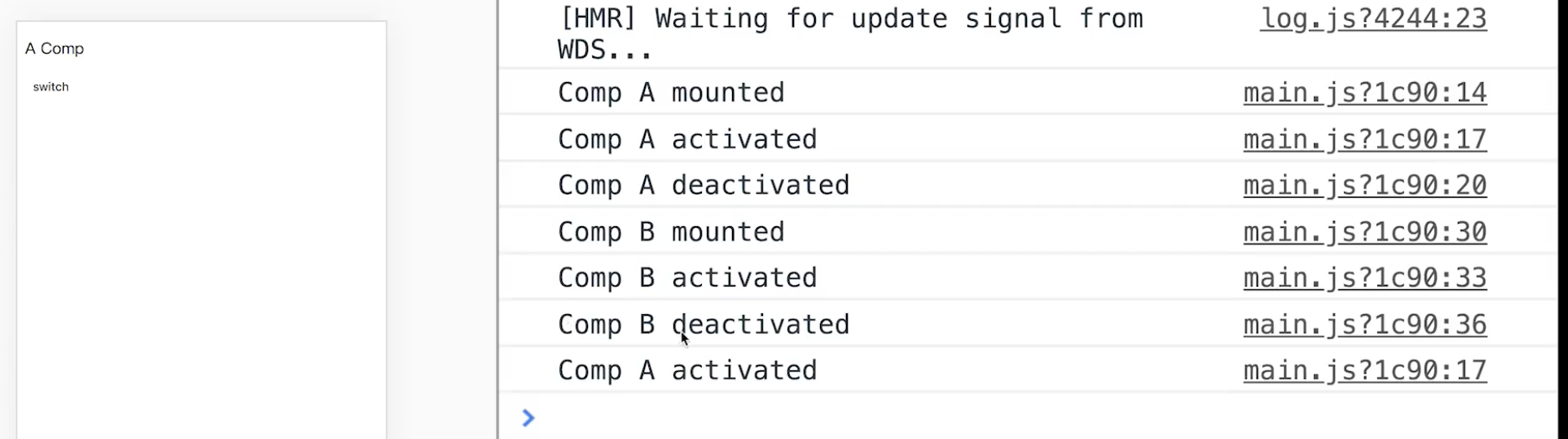
# 扩展（下）

* keep-alive
* 学习目标
* 了解keep-alive的组件实现
* 了解keep-alive渲染机制和生命周期
* 总结
* keep-alive组件是一个内置抽象组件，它的实现通过自定义render函数并且利用了插槽
* keep-alive组件的渲染分为首次渲染和缓存渲染，当命中缓冲，则不会再执行created和mounted钩子函数，而会执行activated钩子函数。销毁时也不会执行destroyed钩子函数，而会执行deactivated钩子函数。









* transition
* 学习目标
* 了解transition的组件实现原理
* 了解transition对过渡动画的控制原理
* 总结
* transition组件是一个内置抽象组件，它的实现通过自定义render函数并利用了插槽
* 真正执行动画的是我们写的CSS或者是Javascrip钩子函数，而transition只是帮我们很好地管理了这些CSS的添加／删除，以及对钩子函数的执行时机
* transition-group
* 学习目标
* 了解transition-group组件实现
* 了解transition-group对过渡动画的控制原理
* 总结
* transition-group组件是为了做列表的过渡，它会渲染成真实的元素
* 当我们去修改列表的数据时候，如果添加或者删除数据，则会触发相应元素本身的过渡动画，这点和<transition>组件实现效果一样，除此之外<transition-group>还实现了move的过渡效果，让我们的列表国度动画更加丰富。

# Vue-Router

* 路由注册
* 学习目标
* 了解Vue插件的注册原理
* 了解路由注册的实现流程
* 总结
* Vue编写插件的时候通常要提供静态的install方法
* Vue-Router的install方法会给每一个组件注入beforeCreated和destoryed钩子函数，在beforeCreated做一些私有属性定义和路由初始化工作。
* VueRoute对象
* 学习目标
* 了解VueRouter对象的一些熟悉和方法；
* 了解VueRouter对象的初始化逻辑
* 总结
* 路由初始化的时机是在组件的初始化阶段，执行到beforeCreate钩子函数的时候会执行router.init方法。然后又会执行history.transitionTo方法做路由过渡。
* matcher
* 学习目标
* 了解createMatcher的初始化逻辑
* 了解match的匹配过程
* 总结
* createMatcher的初始化就是根据路由的匹配描述创建映射表，包括路径、名称到路由record的映射关系。
* match会根据传入的位置和路径计算出新的位置，并匹配到对应的路由record，然后根据新的位置和record创建新的路径返回。
* 路径切换
* 学习目标
* 了解导航守卫的执行逻辑
* 了解url的变化逻辑
* 了解组件的渲染逻辑
* 总结
* 路由始终会维护当前的线路，路由切换的时候会把当前线路切换到目标线路，切换过程会执行一系列的导航首位钩子函数，会更改url，同样也会渲染对应的组件，切换完毕后会把目标线路更新替换当前线路，这样就会座位下一次的路径切换的依据
* 导航守卫

全局前置守卫：router.beforeEach

全局解析守卫：router.beforeResolve

全局后置钩子：router.afterEach

* 组件内导航

beforeRouterEnter:在渲染该组件的对应路由被confirm前调用

beforeRouterUpdate:在当前路由改变，但是该组件被复用时调用

beforeRouterLeave:导航离开该组件的对应路由时调用

* 路由独享的守卫

beforeEnter：在路由配置上直接定义

* 导航守卫执行流程
* 导航被触发。
* 在失活的组件里调用离开守卫。
* 调用全局的 beforeEach 守卫。
* 在重用的组件里调用 beforeRouteUpdate 守卫 (2.2+)。
* 在路由配置里调用 beforeEnter。
* 解析异步路由组件。
* 在被激活的组件里调用 beforeRouteEnter。
* 调用全局的 beforeResolve 守卫 (2.5+)。
* 导航被确认。
* 调用全局的 afterEach 钩子。
* 触发 DOM 更新。
* 用创建好的实例调用 beforeRouteEnter 守卫中传给 next 的回调函数。

# Vuex

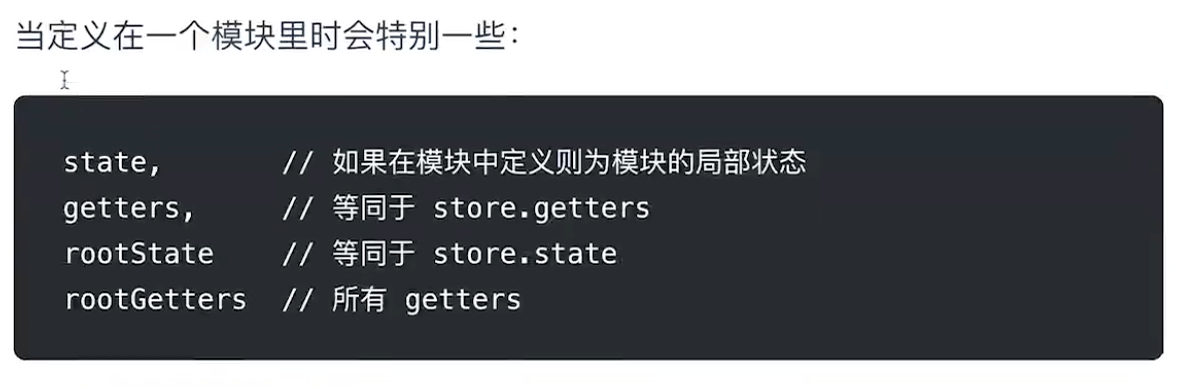


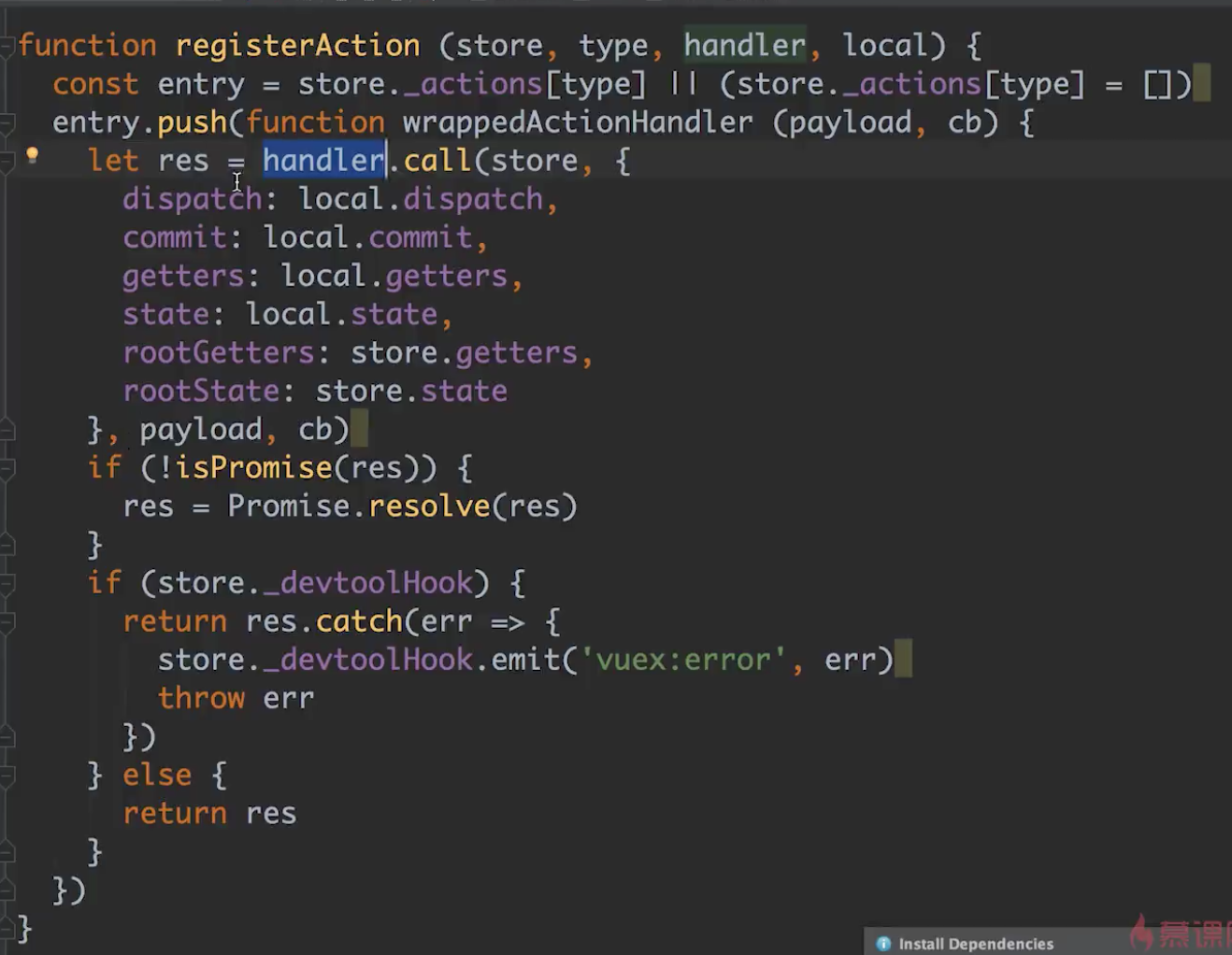
action：



为什么context可以调用commit原因

getter：





* Vuex介绍

Vuex 是一个专为Vue.js应用程序开发的状态管理模式。它采用集中式存储管理应用的所有组件的状态，并以相应的规则保证状态以一种可预测的方式发生变化。

* Vuex初始化
* 学习目标
* 了解Vuex的安装过程
* 了解Store的实力化过程
* 总结
* store就是一个数据仓库，为了更方便的管理仓库，我们把一个大的store拆成一些modules，整个modules是一个树型结构。每个module又分别定义了state、getters、mutations、actions，我们也通过递归遍历模块的方式都完成了它们的初始化。
* API
* 学习目标
* 了解Vuex数据存取的API；
* 了解常用的语法糖
* 了解模块动态注册流程
* 总结
* Vuex提供这些API都方便我们在对store做各种操作来完成各种能力，尤其是mapXXX的设计，让我们在使用API的时候更加方便，着也是我们今后设计一些javascript库的时候，从API设计角度中应该学习的方向。
* 插件
* 学习目标
* 了解Vuex插件的设计
* 了解Logger插件的实现原理
* 总结

Vuex从设计上支持了插件，让我们很好地从外部追踪store内部的变化，logger插件在我们的开发阶段也提供了很好地指引作用。当然我们也可以自己去实现Vuex的插件，来帮助我们实现一些特定的需求。