目录

[1 Vue基础 1](#_Toc90150927)

[1.1Vue组件之间的通信方式 1](#_Toc90150928)

[1.2 Vuex 4](#_Toc90150929)

[1.3 $attrs/$listeners 4](#_Toc90150930)

[1.4 Vue的响应式原理 7](#_Toc90150931)

[1.4.1 Object.defineProperty 7](#_Toc90150932)

[1.4.2 观察者模式 9](#_Toc90150933)

[1.5 MVVM 11](#_Toc90150934)

[1.6 Vue实例是什么意思 11](#_Toc90150935)

[1.7 Vue的双向绑定 12](#_Toc90150936)

[1.8 Vue的虚拟DOM树原理 14](#_Toc90150937)

[1.8.1 传统HTML的DOM树构建过程 14](#_Toc90150938)

[1.8.2 传统HTML下用js操作DOM的代价 14](#_Toc90150939)

[1.8.3 虚拟DOM 15](#_Toc90150940)

[1.9 Vue的生命周期 15](#_Toc90150941)

# Vue基础

## 1.1Vue组件之间的通信方式

·props 和emit 父组件向子组件传递数据是通过 prop 传递的，子组件传递数据给父组件是通过emit 触发事件来做到的。

·parent,children 获取当前组件的父组件和当前组件的子组件

·vuex 状态管理

·attrs 和listeners A->B->C。Vue 2.4 开始提供了attrs 和listeners 来解决这个问题。

方法一：通过使用props和emit的方式进行通信

**1.父组件向子组件传值**

接下来我们通过一个例子，说明父组件如何向子组件传递值：在子组件Users.vue中如何获取父组件App.vue中的数据 users:["Henry","Bucky","Emily"]

//App.vue父组件

<template>

<div id="app">

<users v-bind:users="users"></users>//前者自定义名称便于子组件调用，后者要传递数据名

</div>

</template>

<script>

import Users from "./components/Users"

export default {

name: 'App',

data(){

return{

users:["Henry","Bucky","Emily"]

}

},

components:{

"users":Users

}

}

//users子组件

<template>

<div class="hello">

<ul>

<li v-for="user in users">{{user}}</li>//遍历传递过来的值，然后呈现到页面

</ul>

</div>

</template>

<script>

export default {

name: 'HelloWorld',

props:{

users:{ //这个就是父组件中子标签自定义名字

type:Array,

required:true

}

}

}

</script>

**总结：父组件通过props向下传递数据给子组件。注：组件中的数据共有三种形式：data、props、computed**

**2.子组件向父组件传值（通过事件形式）**

接下来我们通过一个例子，说明子组件如何向父组件传递值：当我们点击“Vue.js Demo”后，子组件向父组件传递值，文字由原来的“传递的是一个值”变成“子向父组件传值”，实现子组件向父组件值的传递。



// 子组件

<template>

<header>

<h1 @click="changeTitle">{{title}}</h1>//绑定一个点击事件

</header>

</template>

<script>

export default {

name: 'app-header',

data() {

return {

title:"Vue.js Demo"

}

},

methods:{

changeTitle() {

this.$emit("titleChanged","子向父组件传值");//自定义事件 传递值“子向父组件传值”

}

}

}

</script>

// 父组件

<template>

<div id="app">

<app-header v-on:titleChanged="updateTitle" ></app-header>//与子组件titleChanged自定义事件保持一致

// updateTitle($event)接受传递过来的文字

<h2>{{title}}</h2>

</div>

</template>

<script>

import Header from "./components/Header"

export default {

name: 'App',

data(){

return{

title:"传递的是一个值"

}

},

methods:{

updateTitle(e){ //声明这个函数

this.title = e;

}

},

components:{

"app-header":Header,

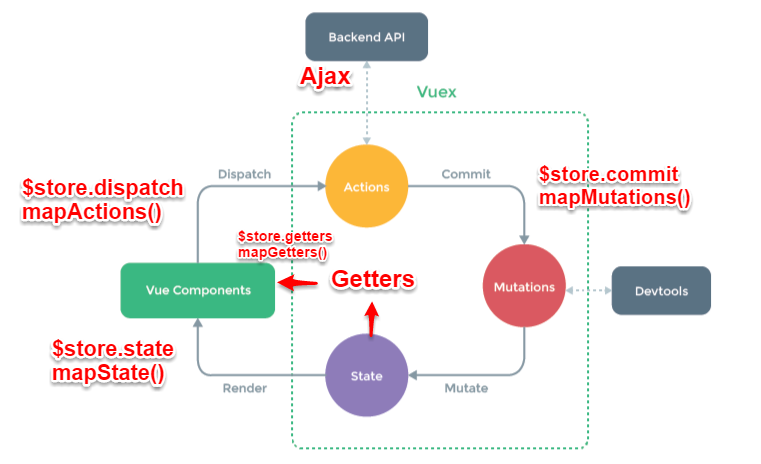
}

}

</script>

**总结：子组件通过events给父组件发送消息，实际上就是子组件把自己的数据发送到父组件。**

## 1.2 Vuex



vuex 是 vue 的状态管理器，存储的数据是响应式的。

## 

## 1.3 $attrs/$listeners

多级组件嵌套需要传递数据时，通常使用的方法是通过vuex。但如果仅仅是传递数据，而不做中间处理，使用 vuex 处理，未免有点大材小用。为此Vue2.4 版本提供了另一种方法----$attrs/$listeners

$attrs：包含了父作用域中不被 prop 所识别 (且获取) 的特性绑定 (class 和 style 除外)。当一个组件没有声明任何 prop 时，这里会包含所有父作用域的绑定 (class 和 style 除外)，并且可以通过 v-bind="$attrs" 传入内部组件。通常配合 interitAttrs 选项一起使用。

$listeners：包含了父作用域中的 (不含 .native 修饰器的) v-on 事件监听器。它可以通过 v-on="$listeners" 传入内部组件

接下来我们看个跨级通信的例子：

// index.vue

<template>

<div>

<h2>浪里行舟</h2>

<child-com1

:foo="foo"

:boo="boo"

:coo="coo"

:doo="doo"

title="前端工匠"

></child-com1>

</div>

</template>

<script>

const childCom1 = () => import("./childCom1.vue");

export default {

components: { childCom1 },

data() {

return {

foo: "Javascript",

boo: "Html",

coo: "CSS",

doo: "Vue"

};

}

};

</script>

// childCom1.vue

<template class="border">

<div>

<p>foo: {{ foo }}</p>

<p>childCom1的$attrs: {{ $attrs }}</p>

<child-com2 v-bind="$attrs"></child-com2>

</div>

</template>

<script>

const childCom2 = () => import("./childCom2.vue");

export default {

components: {

childCom2

},

inheritAttrs: false, // 可以关闭自动挂载到组件根元素上的没有在props声明的属性

props: {

foo: String // foo作为props属性绑定

},

created() {

console.log(this.$attrs); // { "boo": "Html", "coo": "CSS", "doo": "Vue", "title": "前端工匠" }

}

};

</script>

// childCom2.vue

<template>

<div class="border">

<p>boo: {{ boo }}</p>

<p>childCom2: {{ $attrs }}</p>

<child-com3 v-bind="$attrs"></child-com3>

</div>

</template>

<script>

const childCom3 = () => import("./childCom3.vue");

export default {

components: {

childCom3

},

inheritAttrs: false,

props: {

boo: String

},

created() {

console.log(this.$attrs); // {"coo": "CSS", "doo": "Vue", "title": "前端工匠" }

}

};

</script>

// childCom3.vue

<template>

<div class="border">

<p>childCom3: {{ $attrs }}</p>

</div>

</template>

<script>

export default {

props: {

coo: String,

title: String

}

};

</script>

  
如上图所示$attrs表示没有继承数据的对象，格式为{属性名：属性值}。Vue2.4提供了$attrs , $listeners 来传递数据与事件，跨级组件之间的通讯变得更简单。

简单来说：$attrs与$listeners 是两个对象，$attrs 里存放的是父组件中绑定的非 Props 属性，$listeners里存放的是父组件中绑定的非原生事件。

## 1.4 Vue的响应式原理

<https://zhuanlan.zhihu.com/p/53217382> 这篇文章讲的非常清楚 白话文

**响应式原理的定义**：在改变数据的时候，视图会跟着更新。这意味着你只需要进行数据的管理，给我们搬砖提供了很大的便利。

VUE则是利用了Object.defineProperty的方法里面的setter 与getter方法的观察者模式来实现。

### 1.4.1 [Object.defineProperty](https://link.zhihu.com/?target=https%3A//developer.mozilla.org/zh-CN/docs/Web/JavaScript/Reference/Global_Objects/Object/defineProperty)

这个方法就是在一个对象上定义一个新的属性，或者改变一个对象现有的属性，并且返回这个对象。里面有两个字段 set,get。顾名思义，set都是取设置属性的值，而get就是获取属性的值。

举个栗子：

*// 在对象中添加一个属性与存取描述符的示例*

**var** bValue;

**var** o **=** {};

Object.defineProperty(o, "b", {

get **:** **function**(){

console.log('监听正在获取b')

**return** bValue;

},

set **:** **function**(newValue){

console.log('监听正在设置b')

bValue **=** newValue;

},

enumerable **:** **true**,

configurable **:** **true**

});

o.b **=** 38;

console.log(o.b)

最终打印

监听正在设置b

监听正在获取b

38

从在上述栗子中，可以看到当我们对 o.b 赋值38的时候，就会调用[set函数](https://www.zhihu.com/search?q=set%E5%87%BD%E6%95%B0&search_source=Entity&hybrid_search_source=Entity&hybrid_search_extra=%7B%22sourceType%22%3A%22article%22%2C%22sourceId%22%3A88648401%7D" \t "_blank)，这时候给bValue赋值，之后我们就可以通过o.b来获取这个值，这时候，[get函数](https://www.zhihu.com/search?q=get%E5%87%BD%E6%95%B0&search_source=Entity&hybrid_search_source=Entity&hybrid_search_extra=%7B%22sourceType%22%3A%22article%22%2C%22sourceId%22%3A88648401%7D" \t "_blank)被调用。

掌握到这一步，我们已经可以实现一个极简的VUE双向绑定了。

**<**input type**=**"text" id**=**"txt" **/>**

**<**span id**=**"sp"**><**/span>

**<**script**>**

**var** txt **=** document.getElementById('txt'),

sp **=** document.getElementById('sp'),

obj **=** {}

*// 给对象obj添加msg属性，并设置setter访问器*

Object.defineProperty(obj, 'msg', {

*// 设置* [*obj.msg*](https://www.zhihu.com/search?q=obj.msg&search_source=Entity&hybrid_search_source=Entity&hybrid_search_extra=%7B%22sourceType%22%3A%22article%22%2C%22sourceId%22%3A88648401%7D) *当obj.msg反生改变时set方法将会被调用*

set**:** **function** (newVal) {

*// 当obj.msg被赋值时 同时设置给 input/span*

txt.value **=** newVal

sp.innerText **=** newVal

}

})

*// 监听文本框的改变 当文本框输入内容时 改变obj.msg*

txt.addEventListener('keyup', **function** (event) {

obj.msg **=** event.target.value

})

**<**/script>

VUE给data里所有的属性加上set,get这个过程就叫做**Reactive化**。

### 1.4.2 观察者模式

什么是观察者模式？它分为**注册环节跟发布环节**。

比如我去买芝士蛋糕，但是店家还没有做出来。这时候我又不想在店外面傻傻等，我就需要隔一段时间来回来问问蛋糕做好没，对于我来说是很麻烦的事情，说不定我就懒得买了。

店家肯定想要做生意，不想流失我这个吃货客户。于是，在蛋糕没有做好的这段时间，有客户来，他们就让客户把自己的电话留下，这就是观察者模式中的**注册环节**。然后蛋糕做好之后，一次性通知所有记录了的客户，这就是观察者的**发布环节**。

这里来简单实现一个观察者模式的类

**function** Observer() {

**this**.dep **=** [];

register(fn) {

**this**.dep.push(fn)

}

notify() {

**this**.dep.forEach(item => item())

}

}

**const** wantCake **=** **new** Oberver();

*// 每来一个顾客就注册一个想执行的函数*

wantCake.register(() => {'console.log("call daisy")'})

wantCake.register(() => {'console.log("call anny")'})

wantCake.register(() => {'console.log("call sunny")'})

*// 最后蛋糕做好之后，通知所有的客户*

wantCake.notify()

1.4.3 响应式原理

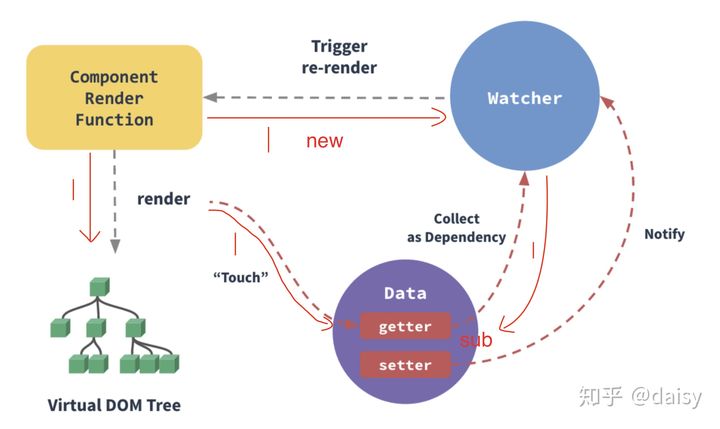
**1、init 阶段：** VUE 的 data的属性都会被reactive化，也就是加上 setter/[getter函数](https://www.zhihu.com/search?q=getter%E5%87%BD%E6%95%B0&search_source=Entity&hybrid_search_source=Entity&hybrid_search_extra=%7B%22sourceType%22%3A%22article%22%2C%22sourceId%22%3A88648401%7D)。

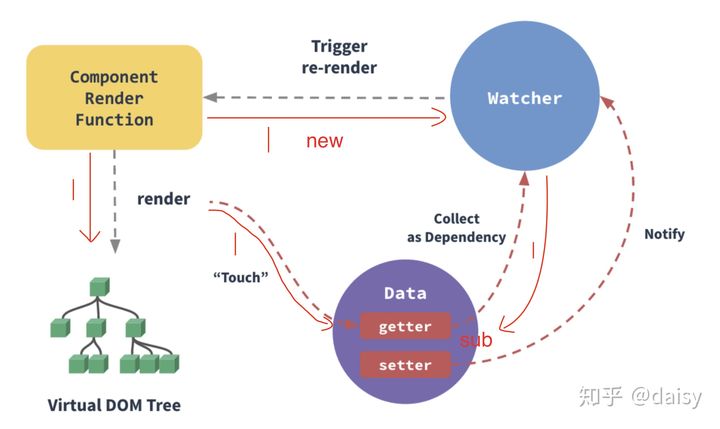
**2、mount 阶段：**mount 阶段的时候，会创建一个Watcher类的对象。这个Watcher实际上是连接Vue组件与Dep的桥梁。  
每一个Watcher对应一个vue component。

**3、更新阶段：**当blogTitle 发生改变的时候，就去调用Dep的[notify函数](https://www.zhihu.com/search?q=notify%E5%87%BD%E6%95%B0&search_source=Entity&hybrid_search_source=Entity&hybrid_search_extra=%7B%22sourceType%22%3A%22article%22%2C%22sourceId%22%3A88648401%7D" \t "_blank),然后通知所有的Watcher调用update函数更新。

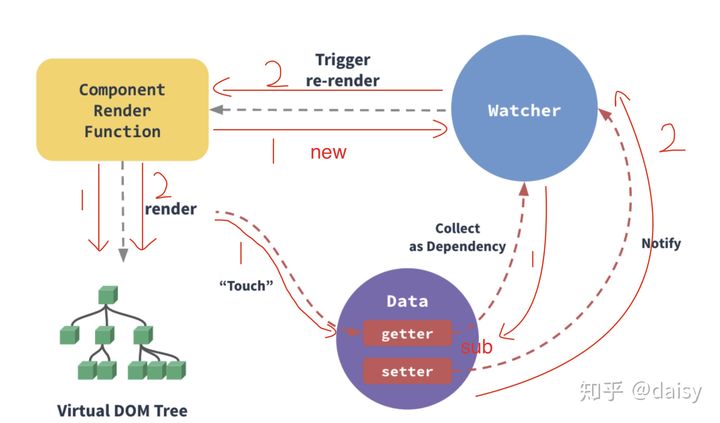
总结

最后，我们通过解释官方的图来做个总结。





**1、第一步：**组件初始化的时候，先给每一个Data属性都注册getter，setter，也就是reactive化。然后再new 一个自己的Watcher对象，此时watcher会立即调用组件的render函数去生成虚拟DOM。在调用render的时候，就会需要用到data的属性值，此时会触发getter函数，将当前的Watcher函数注册进sub里。



**2、第二步：**当data属性发生改变之后，就会遍历sub里所有的watcher对象，通知它们去重新渲染组件。

## 1.5 MVVM



## 1.6 Vue实例是什么意思

Vue实例包含：

类似于创建一个对象,里面包含生命周期钩子函数,data（对象）,methods（方法）.components（计算属性）等...

vue实例的对象data 主要是用来存放数据的 Vue框架会检测data的数据变化,自动更新到html上

methods方法 可以通过vm实例访问这些方法,主要是用来存放函数方法的

计算属性computed 主要是用来一些比较复杂的逻辑计算 虽然{{}} 是非常便利的但是它只能进行哪些比较简单的运算,而且只能够支持单个表达式,多个就会报错。

## 1.7 Vue的双向绑定

双向绑定是视图上的变化能反映到数据上，数据上的变化也能反映到视图上。



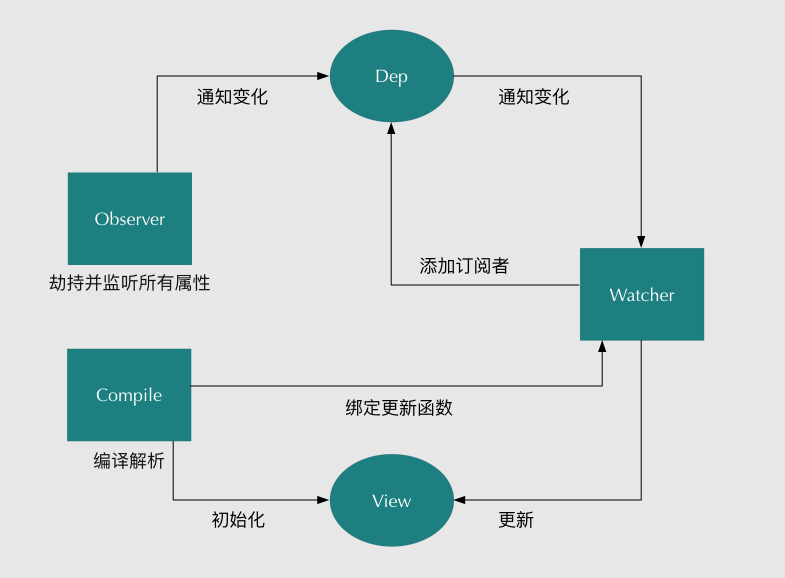
关键点在于data如何更新view，因为view更新data其实可以通过事件监听即可，比如input标签监听 ‘input’ 事件就可以实现了。所以我们着重来分析下，当数据改变，如何更新视图的。

我们已经知道实现数据的双向绑定，首先要对数据进行劫持监听，所以我们需要设置一个监听器Observer，用来监听所有属性。如果属性发上变化了，就需要告诉订阅者Watcher看是否需要更新。因为订阅者是有很多个，所以我们需要有一个消息订阅器Dep来专门收集这些订阅者，然后在监听器Observer和订阅者Watcher之间进行统一管理的。接着，我们还需要有一个指令解析器Compile，对每个节点元素进行扫描和解析，将相关指令对应初始化成一个订阅者Watcher，并替换模板数据或者绑定相应的函数，此时当订阅者Watcher接收到相应属性的变化，就会执行对应的更新函数，从而更新视图。因此接下去我们执行以下3个步骤，实现数据的双向绑定：

1.实现一个监听器Observer，用来劫持并监听所有属性，如果有变动的，就通知订阅者。

2.实现一个订阅者Watcher，可以收到属性的变化通知并执行相应的函数，从而更新视图。

3.实现一个解析器Compile，可以扫描和解析每个节点的相关指令，并根据初始化模板数据以及初始化相应的订阅器。

流程图如下：  


#### Observer

遍历data中的所有属性，同时通过递归的方式遍历这些属性的子属性，通过Object.defineProperty()将这些属性都定义为访问器属性，访问器属性自带get和set方法，我么通过get和set方法来监听数据变化。

#### Dep

我们需要消息订阅器来收集所有的订阅者，就好像是一个列表，当属性的get方法被触发时，需要判断是否要添加订阅者，如果需要就在列表中增加一个订阅者，当set方法被触发时就通知列表中所有的订阅者来做出响应。

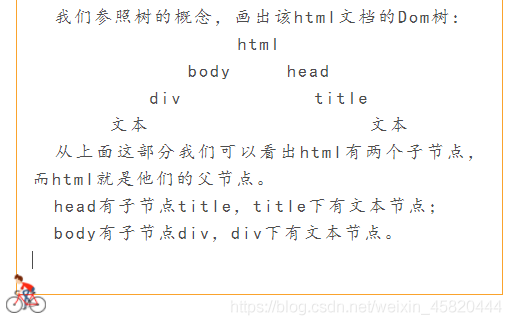
#### Watcher

因为我们是在get函数中判断是否要添加订阅者的，要想把一个订阅者添加到列表中我们就需要在初始化这个订阅者时触发get函数，我们可以在Dep.target上缓存下订阅者，添加成功后再将其去掉就可以了。\

#### Compile

compile负责初始化时的编译解析，遍历每一个结点，看哪些结点需要订阅者，也负责后续为订阅者绑定更新函数。  
OK，这就是我读了博客后自己的总结与理解。

## 1.8 Vue的虚拟DOM树原理



### 1.8.1 传统HTML的DOM树构建过程

浏览器渲染引擎工作流程都差不多，大致分为5步，**创建DOM树——创建StyleRules——创建Render树——布局Layout——绘制Painting**

    第一步，用HTML分析器，分析HTML元素，**构建一颗DOM树**(标记化和树构建)。

    第二步，用CSS分析器，分析CSS文件和元素上的inline样式，生成页面的样式表。

    第三步，将DOM树和样式表，关联起来，构建一颗Render树(这一过程又称为Attachment)。每个DOM节点都有**attach方法，接受样式信息**，返回一个render对象(又名renderer)。这些render对象最终会被构建成一颗Render树。

    第四步，有了Render树，浏览器开始布局，为每个Render树上的节点确定一个在显示屏上出现的精确坐标。

    第五步，Render树和节点显示坐标都有了，就调用每个节点**paint方法，把它们绘制**出来。

**CSS的解析是从右往左逆向解析的**(从DOM树的下－上解析比上－下解析效率高)，**嵌套标签越多，解析越慢。**

### 1.8.2 传统HTML下用js操作DOM的代价

  用我们传统的开发模式，原生JS或JQ操作DOM时，浏览器会从构建DOM树开始从头到尾执行一遍流程。在一次操作中，我需要更新10个DOM节点，浏览器收到第一个DOM请求后并不知道还有9次更新操作，因此会马上执行流程，最终执行10次。例如，第一次计算完，紧接着下一个DOM更新请求，这个节点的坐标值就变了，前一次计算为无用功。计算DOM节点坐标值等都是白白浪费的性能。即使计算机硬件一直在迭代更新，操作DOM的代价仍旧是昂贵的，频繁操作还是会出现页面卡顿，影响用户体验。

### 1.8.3 虚拟DOM

Web界面由DOM树(树的意思是数据结构)来构建，当其中一部分发生变化时，其实就是对应某个DOM节点发生了变化，

        虚拟DOM就是为了**解决浏览器性能问题**而被设计出来的。**如前**，若一次操作中有10次更新DOM的动作，虚拟DOM不会立即操作DOM，而是将这10次更新的diff内容保存到本地一个JS对象中，最终将这个JS对象一次性attch到DOM树上，再进行后续操作，避免大量无谓的计算量。**所以，**用JS对象模拟DOM节点的好处是，页面的更新可以先全部反映在JS对象(虚拟DOM)上，操作内存中的JS对象的速度显然要更快，等更新完成后，再将最终的JS对象映射成真实的DOM，交由浏览器去绘制。

虚拟DOM中，在DOM的状态发生变化时，虚拟DOM会进行Diff运算，来更新只需要被替换的DOM，而不是全部重绘。

在Diff算法中，只平层的比较前后两棵DOM树的节点，没有进行深度的遍历。

1.如果节点类型改变，直接将旧节点卸载，替换为新节点，旧节点包括下面的子节点都将被卸载，如果新节点和旧节点仅仅是类型不同，但下面的所有子节点都一样时，这样做也是效率不高的一个地方。

2.节点类型不变，属性或者属性值改变，不会卸载节点，执行节点更新的操作。

3.文本改变，直接修改文字内容。

4.移动，增加，删除子节点时：





### 1.8.4 **渲染过程中遇到JS文件怎么处理**

JavaScript的加载、解析与执行会阻塞DOM的构建，也就是说，在构建DOM时，HTML解析器若遇到了JavaScript，那么它会暂停构建DOM，将控制权移交给JavaScript引擎，等JavaScript引擎运行完毕，浏览器再从中断的地方恢复DOM构建。

  也就是说，如果你想首屏渲染的越快，就越不应该在首屏就加载 JS 文件，这也是都建议将 script 标签放在 body 标签底部的原因。当然在当下，并不是说 script 标签必须放在底部，因为你可以给 script 标签添加 defer 或者 async 属性（下文会介绍这两者的区别）。

  JS文件不只是阻塞DOM的构建，它会导致CSSOM也阻塞DOM的构建。

  原本DOM和CSSOM的构建是互不影响，井水不犯河水，但是一旦引入了JavaScript，CSSOM也开始阻塞DOM的构建，只有CSSOM构建完毕后，DOM再恢复DOM构建。

## 1.9 Vue的生命周期

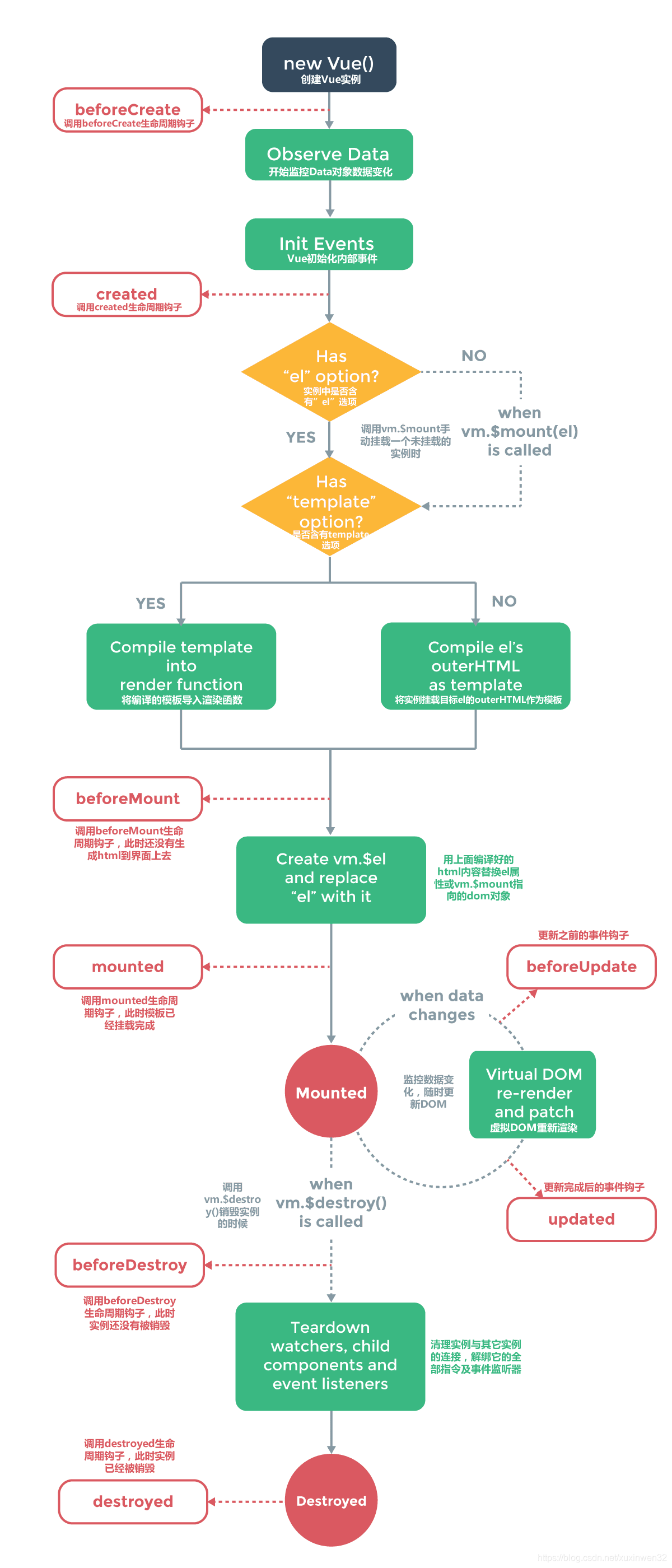
总共分为8个阶段创建前/后，载入前/后，更新前/后，销毁前/后。

**创建前后：**在beforeCreated阶段，vue实例的挂载元素el和数据对象data都为undefined，还未初始化。在created阶段，vue实例的数据对象data有了，el和数据对象data都为undefined，还未初始化。在created阶段，vue实例的数据对象data有了，el还没有。

**载入前后：**在beforeMount阶段，vue实例的$el和data都初始化了，但还是挂载之前为虚拟的dom节点，data.message还未替换。在mounted阶段，vue实例挂载完成，data.message成功渲染。

**更新前后：**当data变化时，会触发beforeUpdate和updated方法。

**销毁前后：**在执行destroy方法后，对data的改变不会再触发周期函数，说明此时vue实例已经解除了事件监听以及和dom的绑定，但是dom结构依然存在。



详细流程：

* new Vue()实例化一个vue实例，然后init初始化event 和 lifecycle， 其实这个过程中分别调用了3个初始化函数（initLifecycle(), initEvents(), initRender()），分别初始化了生命周期，事件以及定义createElement函数，初始化生命周期时，定义了一些属性，比如表示当前状态生命周期状态得\_isMounted ，\_isDestroyed ，\_isBeingDestroyed，表示keep-alive中组件状态的\_inactive，而初始化event时，实际上就是定义了$once、$off、$emit、$on几个函数。而createElement函数是在初始化render时定义的（调用了initRender函数）
* 执行beforeCreate生命周期函数
* beforeCreate执行完后，会开始进行数据初始化，这个过程，会定义data数据，方法以及事件，并且完成数据劫持observe以及给组件实例配置watcher观察者实例。这样，后续当数据发生变化时，才能感知到数据的变化并完成页面的渲染
* 执行created生命周期函数，所以，当这个函数执行的时候，我们已经可以拿到data下的数据以及methods下的方法了，所以在这里，我们可以开始调用方法进行数据请求了
* created执行完后，我们可以看到，这里有个判断，判断当前是否有el参数(这里为什么需要判断，是因为我们后面的操作是会依赖这个el的，后面会详细说)，如果有，我们再看是否有template参数。如果没有el，那么我们会等待调用$mount(el)方法(后面会详细说)。
* 确保有了el后，继续往下走，判断当有template参数时，我们会选择去将template模板转换成render函数（其实在这前面是还有一个判断的，判断当前是否有render函数，如果有的话，则会直接去渲染当前的render函数，如果没有那么我们才开始去查找是否有template模板），如果没有template，那么我们就会直接将获取到的el（也就是我们常见的#app，#app里面可能还会有其他标签）编译成templae, 然后在将这个template转换成render函数。
* 之后再调用beforMount， 也就是说实际从creted到beforeMount之间，最主要的工作就是将模板或者el转换为render函数。并且我们可以看出一点，就是你不管是用el，还是用template, 或者是用我们最常用的.vue文件(如果是.vue文件，他其实是会先编译成为template)，最终他都是会被转换为render函数的。
* beforeMount调用后，我们是不是要开始渲染render函数了，首先我们会先生产一个虚拟dom（用于后续数据发生变化时，新老虚拟dom对比计算），进行保存，然后再开始将render渲染成为真实的dom。渲染成真实dom后，会将渲染出来的真实dom替换掉原来的vm.$el（这一步我们可能不理解，请耐心往下看，后面我会举例说明）,然后再将替换后的$el append到我们的页面内。整个初步流程就算是走完了
* 之后再调用mounted，并将标识生命周期的一个属性\_isMounted 置为true。所以mounted函数内，我们是可以操作dom的，因为这个时候dom已经渲染完成了。
* 再之后，只有当我们状态数据发生变化时,我们在触发beforeUpdate，要开始将我们变化后的数据渲染到页面上了（实际上这里是有个判断的，判断当前的\_isMounted是不是为ture并且\_isDestroyed是不是为false，也就是说，保证dom已经被挂载的情况下，且当前组件并未被销毁，才会走update流程）
* beforeUpdate调用之后，我们又会重新生成一个新的虚拟dom(Vnode)，然后会拿这个最新的Vnode和原来的Vnode去做一个diff算，这里就涉及到一系列的计算，算出最小的更新范围，从而更新render函数中的最新数据，再将更新后的render函数渲染成真实dom。也就完成了我们的数据更新
* 然后再执行updated，所以updated里面也可以操作dom，并拿到最新更新后的dom。不过这里我要插一句话了，mouted和updated的执行，并不会等待所有子组件都被挂载完成后再执行，所以如果你希望所有视图都更新完毕后再做些什么事情，那么你最好在mouted或者updated中加一个$nextTick（），然后把要做的事情放在$netTick()中去做（至于为什么，以后讲到$nextTick再说吧）
* 再之后beforeDestroy没啥说的，实例销毁前，也就是说在这个函数内，你还是可以操作实例的
* 之后会做一系列的销毁动作，解除各种数据引用，移除事件监听，删除组件\_watcher，删除子实例，删除自身self等。同时将实例属性\_isDestroyed置为true
* 销毁完成后，再执行destroyed

## 1.10 Vue每个周期适合哪些场景



## 1.11 为什么说Vue是渐进式的

在我看来，渐进式代表的含义是：主张最少。

每个框架都不可避免会有自己的一些特点，从而会对使用者有一定的要求，这些要求就是主张，主张有强有弱，它的强势程度会影响在业务开发中的使用方式。

比如说，Angular，它两个版本都是强主张的，如果你用它，必须接受以下东西：

* 必须使用它的模块机制- 必须使用它的依赖注入
* 必须使用它的特殊形式定义组件（这一点每个视图框架都有，难以避免）

所以Angular是带有比较强的排它性的，如果你的应用不是从头开始，而是要不断考虑是否跟其他东西集成，这些主张会带来一些困扰。

比如React，它也有一定程度的主张，它的主张主要是函数式编程的理念，比如说，你需要知道什么是副作用，什么是纯函数，如何隔离副作用。它的侵入性看似没有Angular那么强，主要因为它是软性侵入。

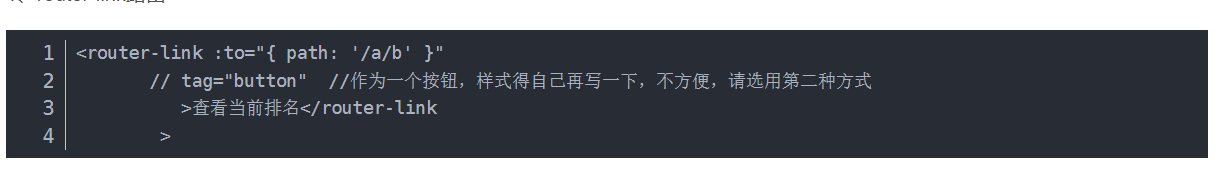
Vue可能有些方面是不如React，不如Angular，但它是渐进的，没有强主张，你可以在原有大系统的上面，把一两个组件改用它实现，当jQuery用；也可以整个用它全家桶开发，当Angular用；还可以用它的视图，搭配你自己设计的整个下层用。你可以在底层数据逻辑的地方用OO和设计模式的那套理念，也可以函数式，都可以，它只是个轻量视图而已，只做了自己该做的事，没有做不该做的事，仅此而已。  
渐进式的含义，我的理解是：没有多做职责之外的事。

## 1.12 Vue渲染的过程

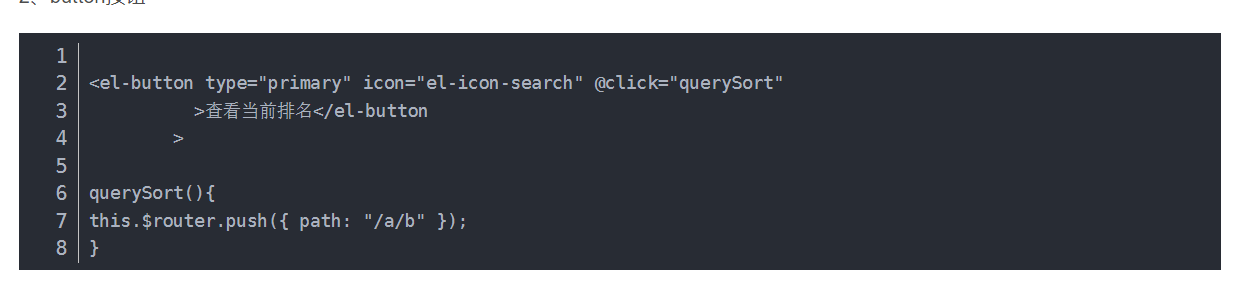
* Vue 推荐在绝大多数情况下使用 template 来创建你的 HTML。但是模板毕竟是模板，不是真实的dom节点。从模板到真实dom节点还需要经过一些步骤
* 把模板编译为render函数
* 实例进行挂载, 根据根节点render函数的调用，递归的生成虚拟dom
* 对比虚拟dom，渲染到真实dom
* 组件内部data发生变化，组件和子组件引用data作为props重新调用render函数，生成虚拟dom, 返回到步骤3

## 1.13 vue页面跳转的方法

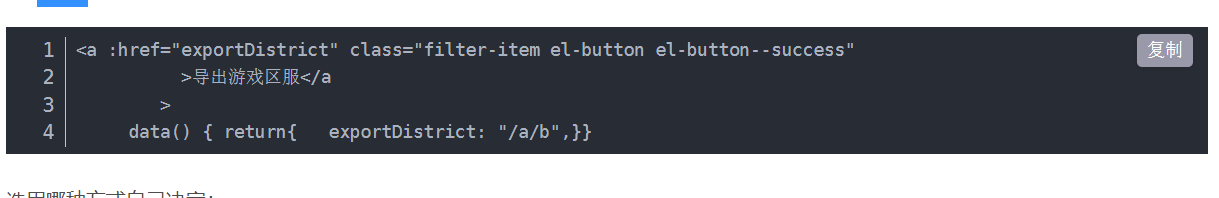
**router-link路由**



**This.$route.push()**



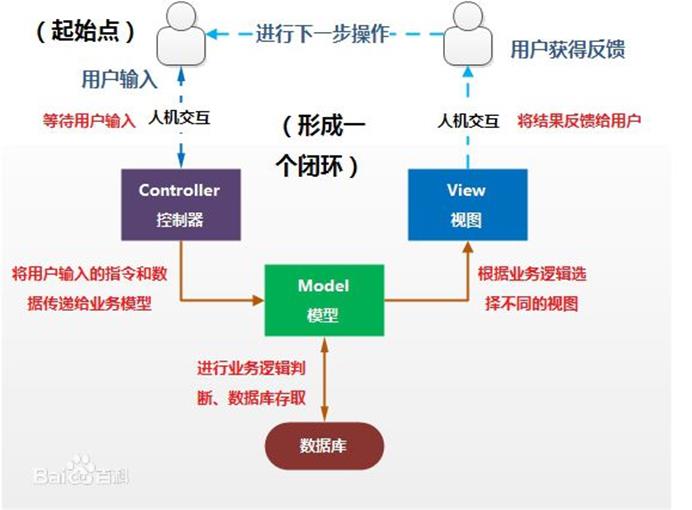
**标签 a 的超链接实现**



# 4 实战篇

## 4.1 字节

4.1.1 MVC



几乎所有的App都只干这么一件事：将数据展示给用户看，并处理用户对界面的操作。MVC的思想：**一句话描述就是Controller负责将Model的数据用View显示出来，换句话说就是在Controller里面把Model的数据赋值给View。**

**View和Controller的交互**

按钮点击事件，是View来接收的，但是处理这个事件的应该是Controller，所以View把这个事件传递给了Controller，如何传递的呢，见图，看到View上面的action没有，这就是事件，看到Controller上面的target没有，这就是靶子，View究竟要把事件传递给谁，它被规定了传递给靶子，Controller实际上就是靶子。只是View只负责传递事件，不负责关心靶子是谁。就像你是一个负责运货的少年，你唯一知道的是你要把货（action）交给上头（开发者）告诉你的那个收货的人（target），至于那个收货的人是警察还是怪兽，你都不需要关心。这是V和C的一种交互方式，叫做target-action。所以你看，这张图简直就是神来之笔，旁边还栩栩如生的画出了V对C的另一种传值：协议-委托。委托有两种：代理和数据源。什么是代理，就是专门处理should、will、did事件的委托，什么是数据源，就是专门处理data、count等等的委托。

**Model和Controller的交互**

M是干嘛的？上面说了，M就是数据管理者，你可以理解为它直接和数据库打交道。这里的数据库可能是本地的，也可能是服务器上的，M会从数据库获取数据，也可能把数据上传给数据库。M也将提供属性或者接口来供C访问其持有的数据。我们就拿一个简单的需求作为例子，假如我想在一个模块中显示一段文字，这段文字是从网上获取下来的。

那么使用MVC的话，在C中肯定需要一个UILabel（V）作为属性来显示这段文字，而这段文字由谁来获取呢，肯定是由M来获取了。而获取的地方在哪里呢？通常在C的生命周期里面，所以往往是在C的一个生命周期方法比如viewDidLoad里面调用M获取数据的方法来获取数据。现在问题来了，M获取数据的方法是异步的网络请求，网络请求结束后，C才应该用请求下来的数据重新赋值给V，现在的问题是，C如何知道网络请求结束了？

这里我们一定要换一种角度去思考，我们进一步考虑M和V之间的关系：它们应该是一种同步的关系，也就是，不管任何时刻，只要M的值发生改变，V的显示就应该发生改变（显示最新的M的内容）。所以我们可以关注M的值改变，而不用关心M的网络请求是否结束了。实际上C根本不知道M从哪去拿的数据，C的责任是负责把M最新的数据赋值给V。所以C应该关注的事件是：M的值是否发生了变化。

### 4.1.2 MVVM

