Vue实例

1 创建实例

```
var vm = new Vue({
  // 选项
})
```

- 每个 Vue 应用都是通过用 Vue 函数创建一个新的 Vue 实例开始的
- 一个 Vue 应用由一个通过 new Vue 创建的根 Vue 实例，以及可选的嵌套的、可复用的组件树组成。

2 数据与方法

2.1 数据

- 当一个 Vue 实例被创建时，它向 Vue 的响应式系统中加入了其 data 对象中能找到的所有的属性。当这些属性的值发生改变时，视图将会产生“响应”，即匹配更新为新的值。
- 只有当实例被创建时 data 中存在的属性才是响应式的
- 如果你知道你会在晚些时候需要一个属性，但是一开始它为空或不存在，那么你仅需要设置一些初始值

2.2 实例方法

Vue 实例还暴露了一些有用的实例属性与方法。它们都有前缀 $，以便与用户定义的属性区分开来

- vm.$el
- vm.$data
- vm.$watch(dataAttr, fn)

3 计算属性和侦听器

3.1 methods

methods用来装载可以调用的函数，你可以直接通过 Vue 实例访问这些方法，或者在指令表达式中使用。方法中的 this 自动绑定为 Vue 实例。

> 注意，不应该使用箭头函数来定义 methods 函数（例如 plus: () => this.a++）。理由是箭头函数绑定了父级作用域的上下文，所以 this 将不会按照期望指向 Vue 实例，this.a 将是 undefined。示例代码如下。

如果你要通过对 DOM 的操作来触发这些函数，那么应该使用 v-on 对操作和事件进行绑定

```
var vm = new Vue({
  data: { a: 1 },
  methods: {
    plus: function () {
      this.a++
    }
  }
})

vm.plus()
vm.a // 2
```

3.2 computed 计算属性

模板内的表达式非常便利，但是设计它们的初衷是用于简单运算的。在模板中放入太多的逻辑会让模板过重且难以维护,这时候应该使用计算属性

```
<div id="example">
  {{ message.split('').reverse().join('') }}
</div>

<!--以下是计算属性的用法-->
<div id="example">
  <p>Original message: "{{ message }}"</p>
  <p>Computed reversed message: "{{ reversedMessage }}"</p>
</div>
```

```
var vm = new Vue({
  el: '#example',
  data: {
    message: 'Hello'
  },
  computed: {
    // 计算属性的 getter
    reversedMessage: function () {
      // `this` 指向 vm 实例
      return this.message.split('').reverse().join('')
    }
  }
})
```

3.3 watch 监听器

虽然计算属性在大多数情况下更合适，但有时也需要一个自定义的侦听器。这就是为什么 Vue 通过 watch 选项提供了一个更通用的方法，来响应数据的变化。当需要在数据变化时执行异步或开销较大的操作时，这个方式是最有用的

```
var vm = new Vue({
    data: {
        question: ''
    },
    watch: {
        // 如果 `question` 发生改变，这个函数就会运行
        question: function (newQuestion, oldQuestion) {
          this.answer = 'Waiting for you to stop typing...'
          this.debouncedGetAnswer()
        }
  },
})
```

3.4 三者区别

它们三者都是以函数为主体，但是它们之间却各有区别。

计算属行与方法

我们可以将同一函数定义为一个方法而不是一个计算属性。两种方式的最终结果确实是完全相同的。然而，不同的是计算属性是基于它们的依赖进行缓存的。计算属性只有在它的相关依赖发生改变时才会重新求值。这就意味着只要 message 还没有发生改变，多次访问 reversedMessage 计算属性会立即返回之前的计算结果，而不必再次执行函数。

相比之下，每当触发重新渲染时，调用方法将总会再次执行函数。

计算属性与侦听属性

- watch擅长处理的场景：一个数据影响多个数据
- computed擅长处理的场景：一个数据受多个数据影响

4 生命周期

4.1 生命周期钩子函数

1.beforeCreate

在实例初始化之后，数据观测（data observer）和 event/watcher 事件配置之前被调用。

2.created

在实例创建完成后被立即调用。在这一步，实例已完成以下的配置：数据观测（data observer）、属性和方法的运算、watch/event 事件回调。然而，挂载阶段还没开始，$el 属性目前不可见。

3.beforeMount

在挂载开始之前被调用，相关的 render 函数将首次被调用。

注意：该钩子在服务器端渲染期间不被调用。

4.mounted

el 被新创建的 vm.el替换，并挂载到实例上去之后调用该钩子。如果root实例挂载了一个文档内元素，当mounted被调用时vm.el 也在文档内。页面渲染完成后初始化的处理都可以放在这里。

注意：mounted 不会承诺所有的子组件也都一起被挂载。

5.beforeUpdate

数据更新时调用，发生在虚拟 DOM 重新渲染和打补丁之前。

你可以在这个钩子中进一步地更改状态，这不会触发附加的重渲染过程。

6.updated

由于数据更改导致的虚拟 DOM 重新渲染和打补丁，在这之后会调用该钩子。

当这个钩子被调用时，组件 DOM 已经更新，所以你现在可以执行依赖于 DOM 的操作。然而在大多数情况下，你应该避免在此期间更改状态。如果要相应状态改变，通常最好使用计算属性或 watcher 取而代之。

注意：updated 不会承诺所有的子组件也都一起被重绘。

7.activated

keep-alive 组件激活时调用。

8.deactivated

keep-alive 组件停用时调用。

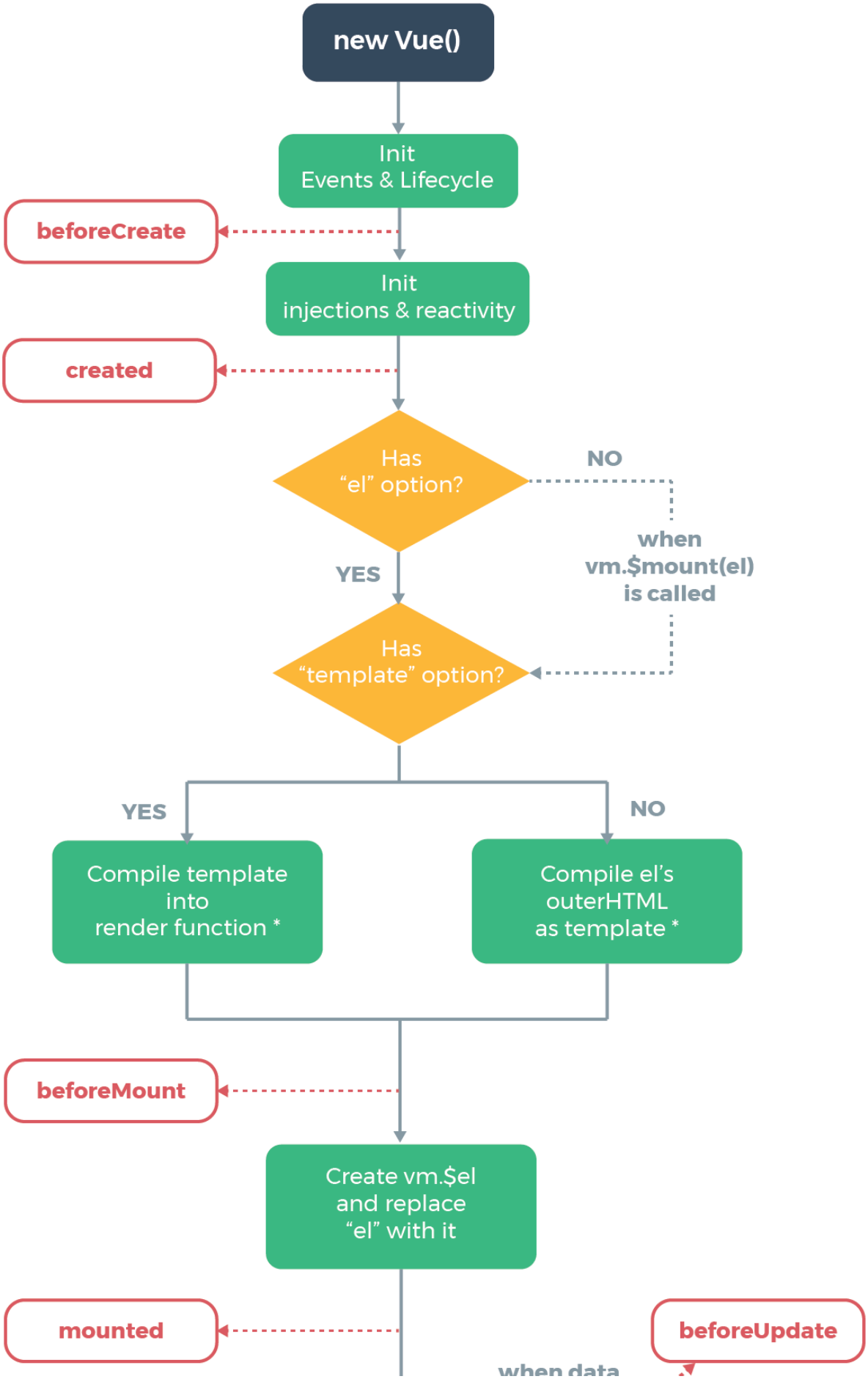
9.beforeDestroy

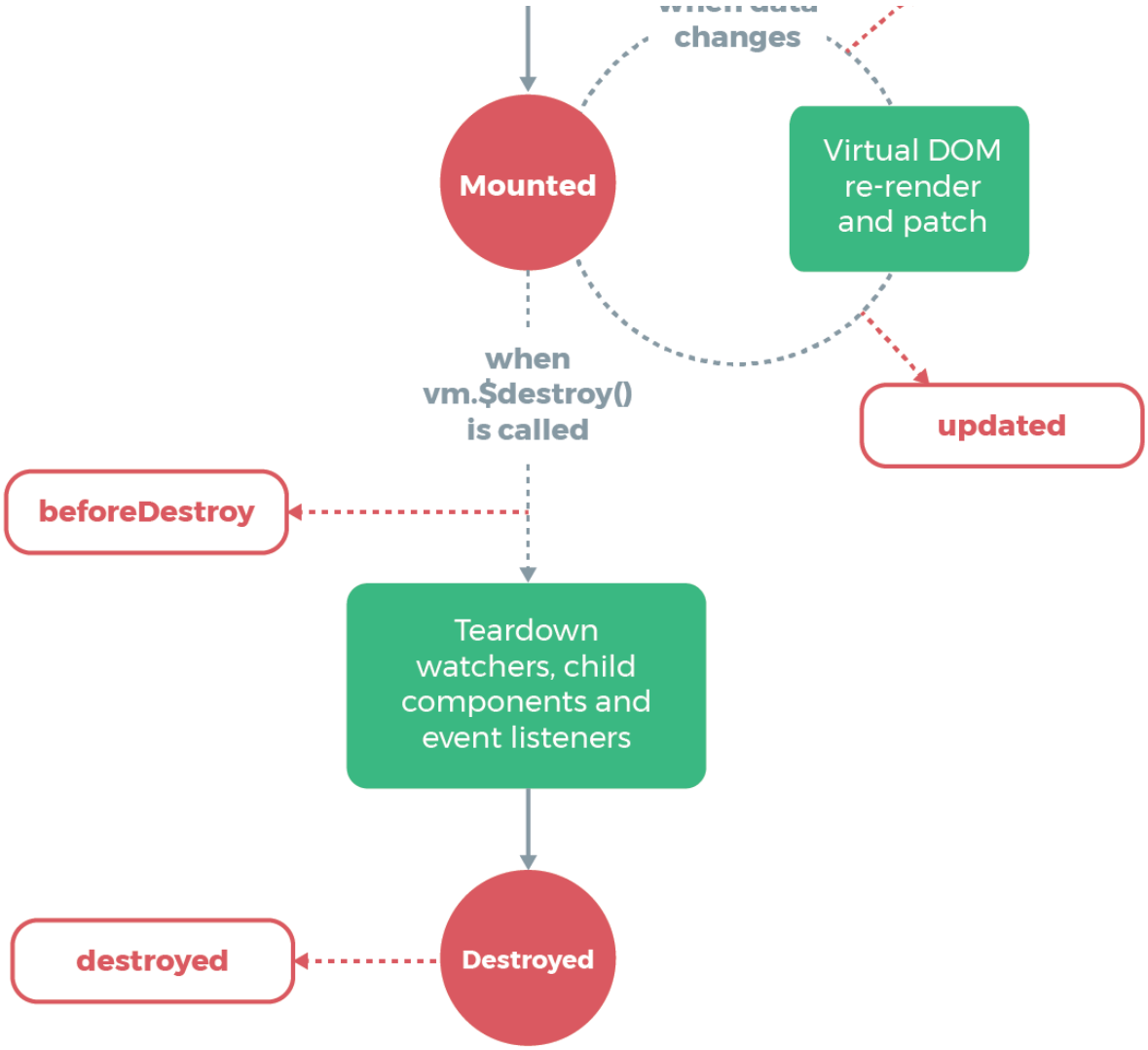
实例销毁之前调用。在这一步，实例仍然完全可用。

10.destroyed

Vue 实例销毁后调用。调用后，Vue 实例指示的所有东西都会解绑定，所有的事件监听器会被移除，所有的子实例也会被销毁。

4.2 生命周期图示


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



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