《前端高频面试100题系列》

一、['1', '2', '3'].map(parseInt) 输出什么?

1、parseInt(str, radix)

- 解析一个字符串，并返回**十进制**整数
- 第一个参数 str，即要解析的字符串
- 第二个参数radix，基数（进制），范围 2-36

2、未传递 radix 参数

- 当 str 以 '0x' 开头，则按照 16 进制处理
- 当 str 以 '0' 开头，则按照 8 进制处理（但 ES5 取消了！！！）
- 其他情况按 10 进制处理

3、分析代码

```
['1',  '2', '3'].map(parseInt)
// 相当于
['1', '2', '3'].map(function(item, index) {
  return parseInt(item, index)
  // parseInt('1', 0) // 注意，这里是按照 radix 参数不存在处理的。
  // parseInt('2', 1)
  // parseInt('3', 2)
})
// 因此结果为：[1, NaN, NaN]
```

二、说出下面代码的执行结果

```
var a = 1
{
  function a() {}
  a = 2
  function a() {}
  a = 3
  function a() {}
  a = 4
  console.log('内部a:', a)
}

console.log('外部a:', a)
```

这个问题可以转化为：书写在**块语句**中的代码是如何预解析的?

我们都知道 JS 中，在 ES6 之前作用域分为 *全局作用域* 和 *函数作用域* ，函数作用域也是 *块作用域* 的一种，虽然这种情况我们平时很少用到，官方API文档也没有说明,但是在实际的过程当中能够发现以下规律：

> 这里所讲的**块作用域**的情况是指 `{ }` 单独应用的时候。

1、当在块语句内引用的时候

- 将函数写在块语句中，命名函数只会预解析，不会预赋值。只有在执行块语句的时候，赋值函数。
- 如果块语句中出现变量和函数名相同的情况时，执行块语句，最后打印的是正常顺序赋值的结果。
- 使用ES6语法 let 和 const 的时候遵从 ES6 对应规则的变量访问机制，定义机制，赋值机制。

2、当在块语句外引用的时候

- 得到的变量，是最后一个**重名函数上面的赋值变量**的结果，如果上面没有重名的赋值变量，那么得到的就是这个函数。
- 在块语句中，不管有几个同名函数，都会被最后一个覆盖掉。

> **总结一句话：**这个只需要记住一点，变量与函数同名时，块作用域中最后一个同名函数上面的变量赋值将会被挤出块外。

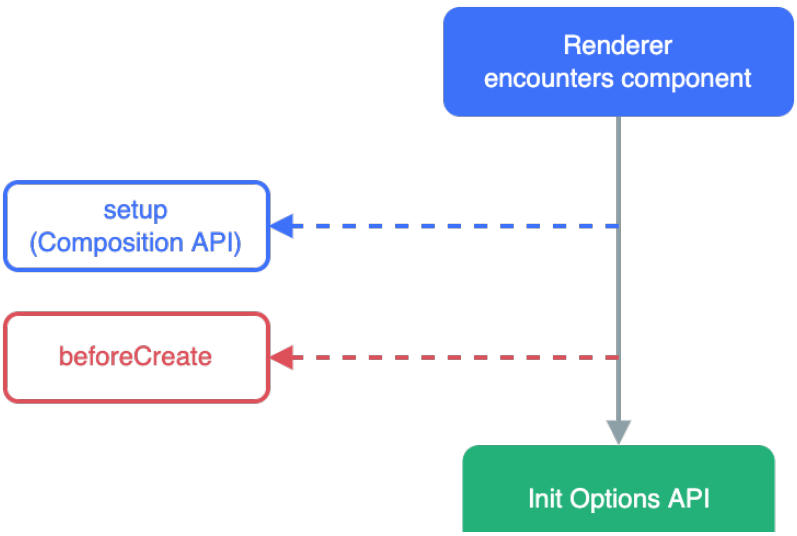
3、分析代码

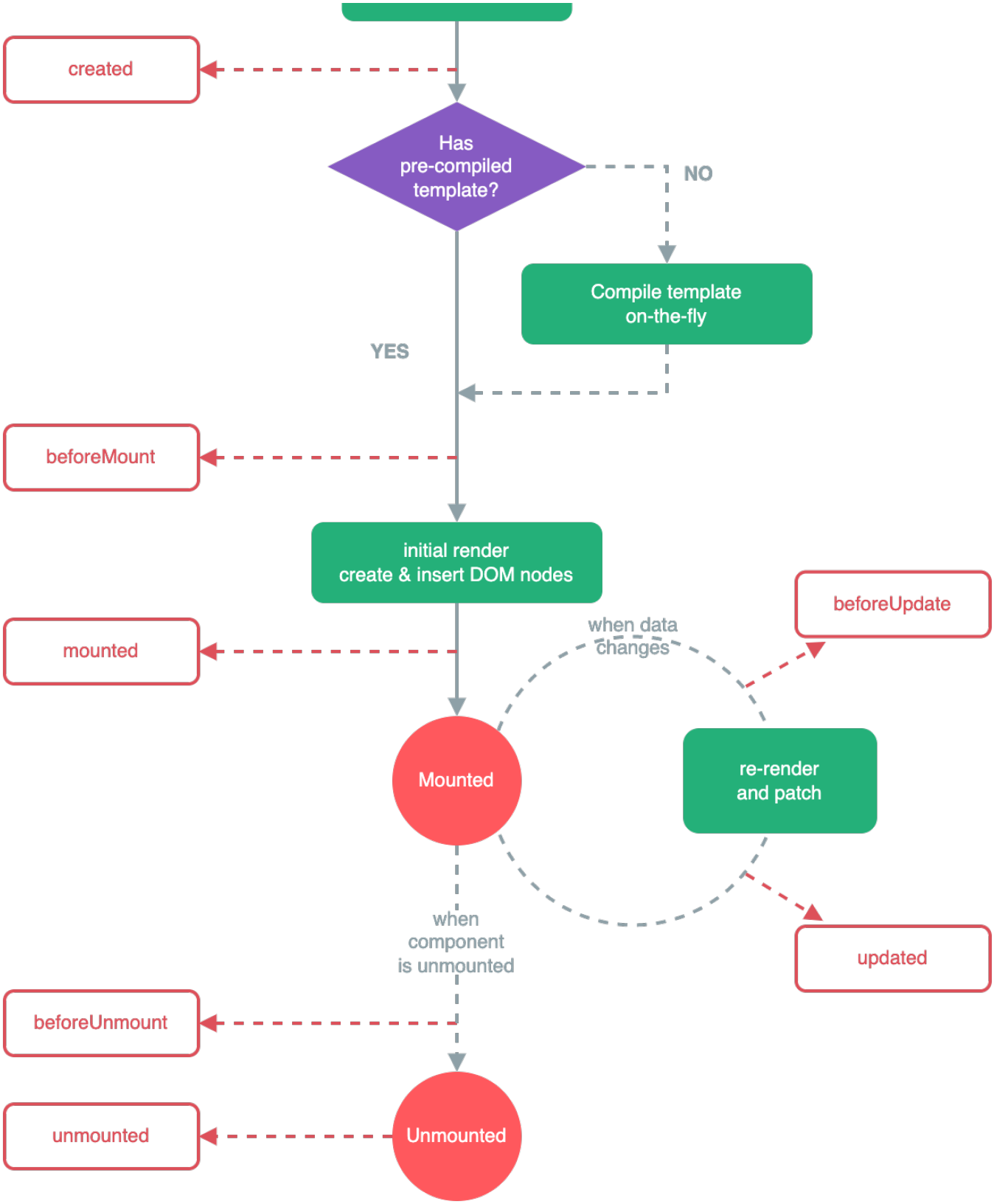
根据以上规律，上述代码的执行结果为：

```
内部a: 4
外部a: 3
```

三、Vue 每个声明周期都做了什么?

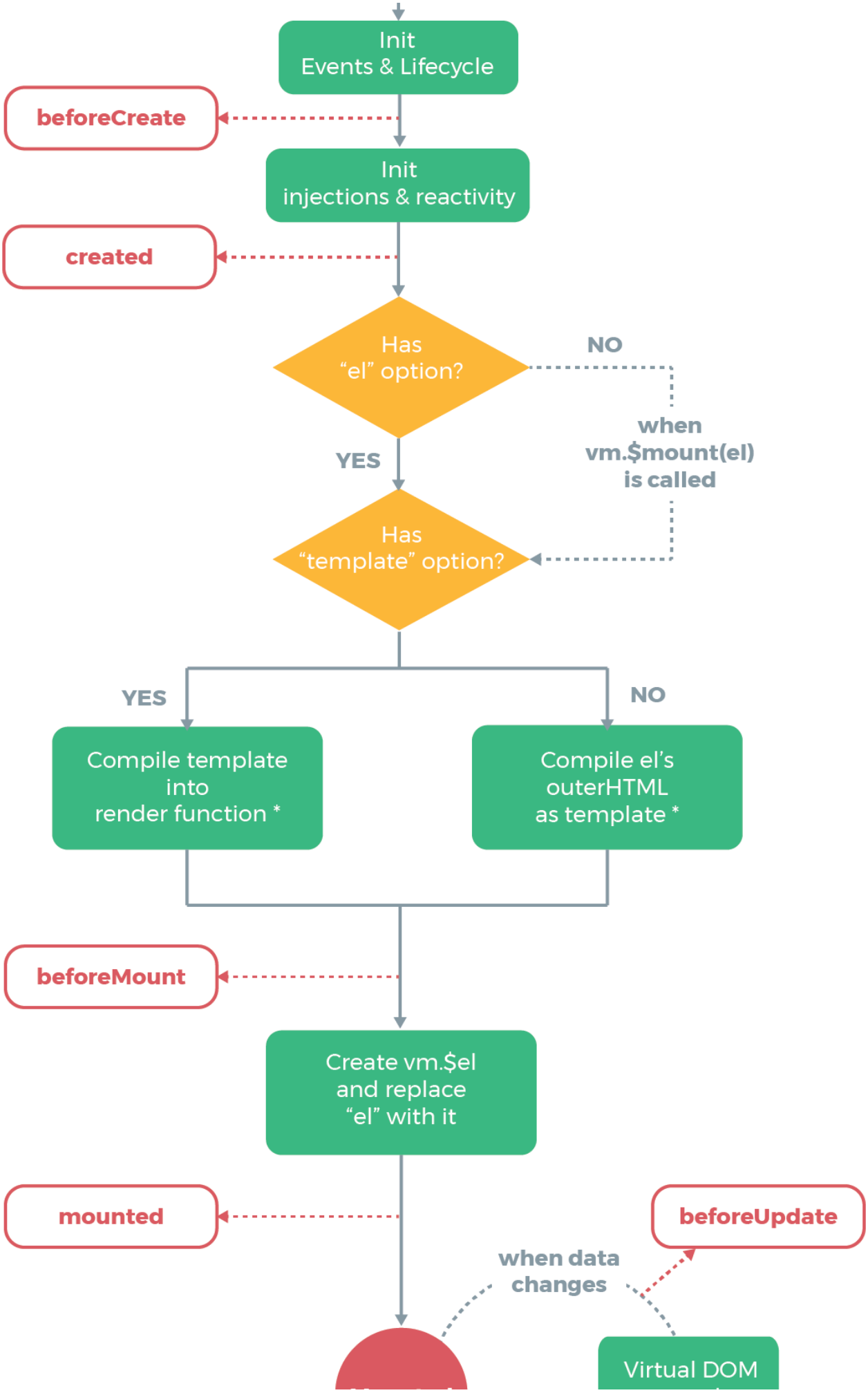
1、Vue3 生命周期

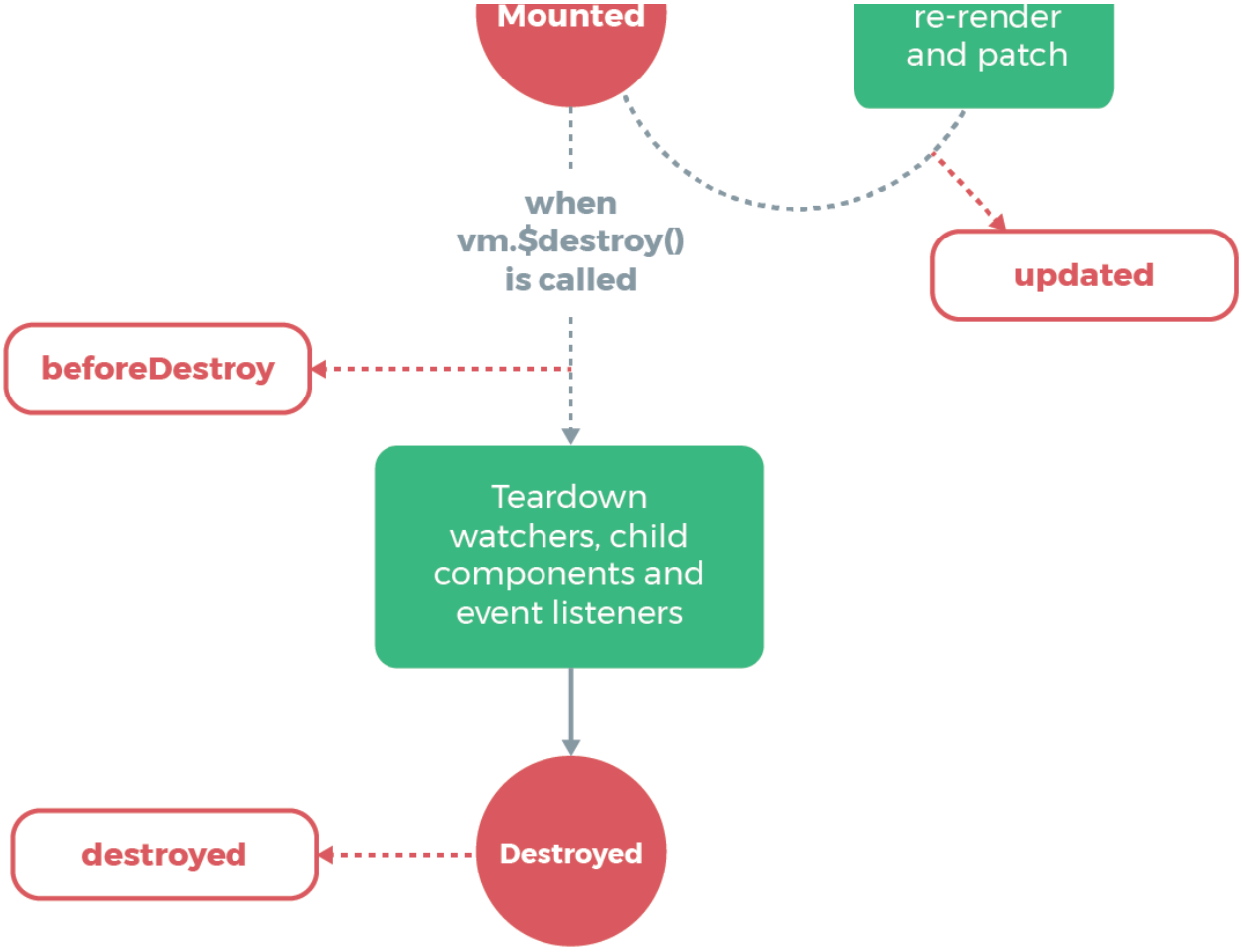



created
Has pre-compiled template?
NO
Compile template on-the-fly
YES
beforeMount
initial render create & insert DOM nodes
mounted
when data changes
beforeUpdate
Mounted
re-render and patch
updated
when component is unmounted
beforeUnmount
unmounted
Unmounted

2、Vue2 生命周期


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
Virtual DOM



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

3、普通钩子说明

（1）beforeCreate

- 创建空白的 Vue 实例
- data、methods 尚未被初始化，不可使用

（2）created

- Vue实例初始化完成，完成响应式绑定
- data、methods 都已经初始化完成，可调用
- 尚未开始渲染模板

（3）beforeMount

- 编译模板，调用 render 生成 vdom
- 还没有开始渲染 DOM

（4）mounted

- 完成 DOM 渲染
- 组件创建完成
- 开始由“创建阶段”进入“运行阶段”

（5）beforeUpdate

- data 发生了变化之后
- 准备更新DOM（尚未更新 DOM）

（6）updated

- data 发生变化，且 DOM 更新完成
- （不要在 updated 中修改 data，可能会导致死循环）

（7）beforeUnmount(beforeDestroy)

- 组件进入销毁阶段（尚未销毁，可正常使用）
- 可移除、解绑一些全局事件、自定义事件

（8）unmounted(destroyed)

- 组件被销毁了
- 所有的子组件也都被销毁了

4、keep-alive 组件钩子说明

- **onActivated(activated)** 缓存组件被激活
- **onDeactivated(deactivated)** 缓存组件被隐藏

四、Vue 什么时候操作 DOM 比较合适?

- 在 mounted 和 updated 都不能保证子组件全部挂载完成
- 使用 $nextTick 渲染 DOM

```
mounted() {
  this.$nextTick(function(){
    // 仅在整个视图被渲染之后才会运行的代码
  })
}
// 或
async mounted() {
  await this.$nextTick()
  // 仅在整个视图被渲染之后才会运行的代码
}
```

五、Ajax 应该在哪个生命周期?

- 有两个选择：created 和 mounted
- 推荐 mounted

六、Vue3 Composition API 生命周期有何区别?

- 用 setup 代替了 beforeCreate 和 created
- 使用 Hooks 函数的形式，如 mounted 改为 onMounted()

七、为什么使用Hooks?

1、回顾 React 复用组件的方法

React 没有提供将可复用性行为“附加”到组件的途径（例如，把组件连接到 store）。如果你使用过 React 一段时间，你也许会熟悉一些解决此类问题的方案，比如 **render props** 和 **高阶组件**。

高阶组件(hoc)

```
// addProps.jsx
const addProps = (Comp) => {
  return (props) => {
    return (
      <Comp {...props} title="hello"></Comp>
    )
  }
}

export default addProps
```

```
// hocComponent.jsx

import { Component } from 'react'

import addProps from './addProps'

class HocComponent extends Component {
  render () {
    return (
      <div>{this.props.title} - {this.props.name}</div>
    )
  }
}

export default addProps(HocComponent)
```

高阶组件的缺点：

- 难以溯源。如果原始组件A通过好几个HOC的构造，最终生成了组件B，不知道哪个属性来自于哪个HOC，需要翻看每个HOC才知道各自做了什么事情，使用了什么属性。
- props属性名的冲突。某个属性可能被多个HOC重复使用。
- 静态构建。新的组件是在页面构建之前生成，先有组件，后生成页面。

渲染属性(render props)

```
// Parent.jsx

import React, { Component } from 'react'
import Child from './Child'

export default class Parent extends Component {
  state = {
    name: 'gp20'
  }

  renderFun = (title) => {
    return (
      <div>{title} {this.state.name}</div>
    )
  }

  render() {
    return (
      <Child render={this.renderFun}></Child>
    )
  }
}
```

```
// Child.jsx

import React, { Component } from 'react'

export default class Child extends Component {
  state = {
    title: 'hello'
  }

  render() {
    return (
      <>
        {this.props.render(this.state.title)}
      </>
    )
  }
}
```

相对高阶组件的优点：

- 不用担心props的命名冲突的问题
- 可以溯源，子组件的props一定来自父组件。
- 是动态构建的，页面在渲染后，可以动态地决定渲染哪个组件。
- 所有能用HOC完成的事情，Render Props都可以做，且更加灵活。
- 除了功能复用，还可以用作两个组件的单向数据传递。

渲染函数和高阶组件共同的缺点：

- 需要重新组织你的组件结构，这可能会很麻烦，使你的代码难以理解。

2、class 组件的问题

除了代码复用和代码管理会遇到困难外，我们还发现 class 是学习 React 的一大屏障。你必须去理解 JavaScript 中 `this` 的工作方式，这与其他语言存在巨大差异。还不能忘记绑定事件处理器。如果不使用 ES2022 public class fields，这些代码非常冗余。大家可以很好地理解 props，state 和自顶向下的数据流，但对 class 却一筹莫展。即便在有经验的 React 开发者之间，对于函数组件与 class 组件的差异也存在分歧，甚至还要区分两种组件的使用场景。

```
// ClassCounter.jsx

import { Component } from 'react'

class Counter extends Component {
  state = {
    counter: 0
  }

  handleClick = () => {
    this.setState((state) => ({
      counter: state.counter + 1
    }))
  }

  render () {
    return (
      <>
        <div>{this.state.counter}</div>
        <button onClick={this.handleClick}>add</button>
      </>
    )
  }
}

export default Counter
```

3、逻辑是分散的，并且难以复用

这个痛点其实跟 `vue2` 是如出一辙的， `React` 的类组件和 `vue2` 的开发模式都是相似。

4、用三个例子演示使用 Hooks 的优点

- **react context**
  - useContext 代替 Consumer
- **react-redux**
  - useSelector、useDispatch 代替 connect
- **逻辑拆分和复用**

- 自定义Hooks

八、React 有哪些Hooks？常用的有哪些？说一下你知道的Hooks的用法和使用场景？

1、React 有哪些 hooks?

Hook 是 React 16.8 的新增特性。它可以让你在不编写 class 的情况下使用 state 以及其他的 React 特性。

- 基础 Hook
  - `useState`
  - `useEffect`
  - `useContext`
- 额外的 Hook
  - `useReducer`
  - `useCallback`
  - `useMemo`
  - `useRef`
  - `useImperativeHandle`
  - `useLayoutEffect`
  - `useDebugValue`
  - `useDeferredValue`
  - `useTransition`
  - `useId`
- Library Hooks
  - `useSyncExternalStore`
  - `useInsertionEffect`

2、常用的有哪些?

useState、useEffect

3、参考文档

《React18-hooks学习指南》

九、Hooks API 能完全代替类组件吗?

这个题目也可以问：Hooks API有什么问题吗?

十、为什么React的Hooks不能在条件语句和循环语句中使用?

道理非常简单。

假设 React 允许 useState 处于条件语句中，那么我们从 React 的视角来看看会发生什么。

假设第一次 render 的 useState 情况如下：

```
const [a, setNextA] = useState(0)
const [b, setNextB] = useState(0)
```

此时用户触发 setNextB(b+1) 于是有了第二次 render。

假设第二次 render 的 useState 情况如下：

```
const [b, setNextB] = useState(0)
```

即，由于某种原因，你少调用了一次 useState。

请问，此时 b 等于多少？你会说 b 加 1 了呀，当然变成 1 了，但是 React 怎么会知道这些?

React 只知道

1. 第一次 render 时 useState 被调用了 2 次，记为调用 1 和调用 2
2. 调用 2 返回的数组的第二项被调用了，即 setNextB(b+1)，触发第二次 render
3. 第二次 render 时 useState 被调用了 1 次，记为调用 3

请问，调用 3 返回的结果是要跟调用 1 相关还是要跟调用 2 相关?

答案是：React 不知道。

因为从 React 角度来看，根本就看不见变量 a 和变量 b！React 只知道 useState 的返回值被你拿去了，但是你拿去之后赋值给什么变量，React 根本就不知道。

如果你一定想让 React 识别 a 和 b，那么可以用下面假想的 useState 把 a 和 b 传给 React：

```
const [a, setNextA] = useState('a', 0) // 这种 useState 是不存在的
const [b, setNextB] = useState('b', 0)
```

加上 key 之后，后续的 render 就能匹配上 a 和 b 了。

因为你的第二次 render 是这样的：

```
const [b, setNextB] = useState('b', 0)
```

React 当然就会知道这是 b 不是 a。

相信你已经发现问题所在了，React 不允许 hook 处于条件语句是因为 React 把每次 render 中 useState 的顺序值 0、1、2、3 当成了 key，方便后续 render 用 key 查找对应的 state。这样的目的是使 useState 更简洁。（不止 useState，其他 hook 也不允许处于条件语句中）

当然，React 这么做的目的也可能是为了追求函数式或者并发，但显然大部分前端开发者并不在乎这些。

注意，并不是因为 hooks 内部使用链表来实现，所以我们必须保证 hooks 的调用顺序。这种观点显然倒置了因果关系，正确的说法是：因为我们保证了 hooks 的调用顺序（不保证就会报错），所以 hooks 内部可以使用链表来实现。

十一、React Hooks 怎么避免不必要的更新?

十二、类组件可以对Props进行前后对比，Hooks 是怎么对比的?

十三、Vue3的组合式Api为什么能在条件和循环语句中使用?

因为 Vue 3 的 setup 并不是一个常规函数，而是含有一个闭包（**闭包 = 自由变量 + 函数**）的函数。代码如下：

```
export const Fang = defineComponent({
  props: ...,
  setup: ()⇒{
    const a = ref(0)
    const b = ref(0)
    return () ⇒ ( // 此箭头函数记为 fnReturn
      <div>{a.value} {b.value}</div>
    )
  }
})
```

此处使用 JSX 语法，以方便与 React 做对比。

当你用 b.value += 1 触发 re-render 时，并不会重新执行 setup 函数，只会重新执行fnReturn，因此 Vue 根本就不需要找 a 和 b（想找也找不到，原因跟 React 一样，Vue 只知道你拿走了 ref(0) 的返回值），因为 a 和 b 都是 fnReturn 可以直接读到的自由变量啊！换句话说，a、b、fnReturn 三者组成了闭包，这个闭包会一直维持着 a 和 b 变量，提供给 fnReturn 访问。

Vue 3 只需要调用 fnReturn 即可。

理论上，你甚至可以把 a = ref(0) 写在 setup 外面！就像这样：

```
const a = ref(0)
const b = ref(0)
export const Fang = defineComponent({
  props: ...,
  setup: ()=>{
    return () => (  // fnReturn
      <div>{a.value} {b.value}</div>
    )
  }
})
```

这样写完全没有问题，因为 fnReturn 依然可以访问 a 和 b。

所以你也完全可以把 a 放在 if 里，只要 a 能被 fnReturn 访问就行：

```
export const Fang = defineComponent({
  props: ...,
  setup: ()=>{
    let a
    if(window.name === 'Fang'){
      a = ref(0)
    }
    const b = ref(0)
    return () => (
      <div>
        {a && a.value} a 有可能是 undefined，所以要判断一下
        {b.value}
      </div>
    )
  }
})
```

此时你可能会想，Vue 3 这么方便肯定会有什么缺点吧？总不会完爆 React 吧？

其实缺点也有，那就是会让代码里出现很多 .value。还好 SFC 可以帮我们减少 template 里的 .value。

你说，Vue 3 如果能做到用 b += 1 来改变 b 的值不就不用写 .value 了？

但问题在于 JS 并没有提供这种「监听变量被重新赋值」的能力给开发者，所以 Vue 3 做不到这一点。JS 只提供了「监听对象的属性被重新赋值」的能力，所以 Vue 3 可以监听对象 b 的 value 是否被重新赋值。