**我对 React V16.4 生命周期的理解**

原创 现代前端教程 [技术漫谈](javascript:void(0);) 6月27日

来自专辑

深入浅出 React 技术栈

网上有很多关于 React 生命周期的文章，我也看了不少，为了梳理并加深我对此的理解，所以决定写这篇文章。本文主要梳理目前最新的 V16.4 的生命周期函数。本文参考了一些文章，在文末给出了出处，推荐阅读。

先上示意图：

React 生命周期示意图

**废弃三个旧的生命周期函数**

React V16.3 中，为下面三个生命周期函数加上了 UNSAFE 前缀：

* UNSAFE\_componentWillMount
* UNSAFE\_componentWillReceiveProps
* UNSAFE\_componentWillUpdate

React 为三个生命周期函数加上了 UNSAFE 前缀，并明确表示会在 V17.0 版本中删除这三个生命周期函数。先来说说 React 为什么要这么做。

主要是这些生命周期方法经常被误用和滥用。并且在 React V16.0 之前，React 是同步渲染的，而在 V16.0 之后 React 更新了其渲染机制，是通过异步的方式进行渲染的，在 render函数之前的所有函数都有可能被执行多次。

长期以来，原有的生命周期函数总是会诱惑开发者在 render 之前的生命周期函数中做一些动作，现在这些动作还放在这些函数中的话，有可能会被调用多次，这肯定不是我们想要的结果。

**废弃 UNSAFE\_componentWillMount 的原因**

有一个常见的问题，有人问为什么不在 UNSAFE\_componentWillMount 中写 AJAX 获取数据的功能，他们的观点是，UNSAFE\_componentWillMount 在 render 之前执行，早一点执行早得到结果。但是要知道，在 UNSAFE\_componentWillMount 中发起 AJAX 请求，不管多快得到结果也赶不上首次 render，数据都是要在 render 后才能到达。

而且 UNSAFE\_componentWillMount 在服务器端渲染也会被调用到（此方法是服务端渲染唯一会调用的生命周期函数），你肯定不希望 AJAX 请求被执行多次，所以这样的 IO 操作放在 componentDidMount 中更合适。

尤其是在 Fiber 启用了异步渲染之后，更没有理由在 UNSAFE\_componentWillMount 中进行 AJAX 请求了，因为 UNSAFE\_componentWillMount 可能会被调用多次，谁也不会希望无谓地多次调用 AJAX 吧。

还有人会将事件监听器（或订阅）添加到 UNSAFE\_componentWillMount 中，但这可能导致服务器渲染（永远不会调用 componentWillUnmount）和异步渲染（在渲染完成之前可能被中断，导致不调用 componentWillUnmount）的内存泄漏。React Docs

人们通常认为 UNSAFE\_componentWillMount 和 componentWillUnmount 是成对出现的，但这并不能保证。只有调用了 componentDidMount 之后，React 才能保证稍后调用componentWillUnmount 进行清理。因此，添加监听器/订阅的推荐方法是使用componentDidMount 生命周期。React Docs

**废弃 UNSAFE\_componentWillReceiveProps 的原因**

有时候组件在 props 发生变化时会产生副作用。与 UNSAFE\_componentWillUpdate 类似，UNSAFE\_componentWillReceiveProps 可能在一次更新中被多次调用。因此，避免在此方法中产生副作用非常重要。相反，应该使用 componentDidUpdate，因为它保证每次更新只调用一次。

UNSAFE\_componentWillReceiveProps 是考虑到因为父组件引发渲染可能要根据 props 更新state 的需要而设立的。新的 getDerivedStateFromProps 实际上与 componentDidUpdate一起取代了以前的 UNSAFE\_componentWillReceiveProps 函数。

**废弃 UNSAFE\_componentWillUpdate 的原因**

有些人使用 UNSAFE\_componentWillUpdate 是出于一种错误的担心，认为当 componentDidUpdate 触发时，更新其他组件的 state 已经”太晚”了。事实并非如此。React 可确保在用户看到更新的 UI 之前，刷新在 componentDidMount 和 componentDidUpdate 期间发生的任何setState 调用。

通常，最好避免这样的级联更新。当然在某些情况下，这些更新也是必需的（例如：如果你需要在测量渲染的 DOM 元素后，定位工具的提示）。不管怎样，在异步模式下使用 UNSAFE\_componentWillUpdate 都是不安全的，因为外部回调可能会在一次更新中被多次调用。相反，应该使用 componentDidUpdate 生命周期，因为它保证每次更新只调用一次。React Docs

大多数开发者使用 UNSAFE\_componentWillUpdate 的场景是配合 componentDidUpdate，分别获取rerender 前后的视图状态，进行必要的处理。但随着 React 新的 suspense、time slicing、异步渲染等机制的到来，render 过程可以被分割成多次完成，还可以被暂停甚至回溯，这导致UNSAFE\_componentWillUpdate 和 componentDidUpdate 执行前后可能会间隔很长时间，足够使用户进行交互操作更改当前组件的状态，这样可能会导致难以追踪的 BUG。青杨 https://zhuanlan.zhihu.com/p/42413419

React 新增的 getSnapshotBeforeUpdate 方法就是为了解决上述问题，因为 getSnapshotBeforeUpdate 方法是在 UNSAFE\_componentWillUpdate 后（此方法会在 React V17.0 中被正式移除），在 React 真正更改 DOM 前调用的，它获取到组件状态信息更加可靠。

除此之外，getSnapshotBeforeUpdate 还有一个十分明显的好处：它调用的结果会作为第三个参数传入 componentDidUpdate，避免了 UNSAFE\_componentWillUpdate 和 componentDidUpdate 配合使用时将组件临时的状态数据存在组件实例上浪费内存，getSnapshotBeforeUpdate 返回的数据在 componentDidUpdate 中用完即被销毁，效率更高。青杨 https://zhuanlan.zhihu.com/p/42413419

更多问题详见：

* **异步渲染之更新 — React Docs[1]**
* **Update on Async Rendering — React Docs[2]**

**新增两个生命周期函数**

在废弃三个旧的生命周期函数的同时，React 还新增了两个生命周期函数：

* static getDerivedStateFromProps
* getSnapshotBeforeUpdate

**React 生命周期梳理**

React 生命周期主要分为三个阶段：

* 挂载阶段
* 更新阶段
* 卸载阶段

**挂载阶段**

挂载阶段也可以理解为初始化阶段，也就是把我们的组件插入到 DOM 中。这个阶段的过程如下：

* constructor
* getDerivedStateFromProps
* ~~UNSAVE\_componentWillMount~~
* render
* (React Updates DOM and refs)
* componentDidMount

**constructor**

组件的构造函数，第一个被执行。如果在组件中没有显示定义它，则会拥有一个默认的构造函数。如果我们显示定义构造函数，则必须在构造函数第一行执行 super(props)，否则我们无法在构造函数里拿到 this，这些都属于 ES6 的知识。

在构造函数中，我们一般会做两件事：

* 初始化 state
* 对自定义方法进行 this 的绑定

constructor(props) {  
    super(props);  
  
    this.state = {  
      width,  
      height: 'atuo',  
    }  
  
    this.handleChange1 = this.handleChange1.bind(this);  
    this.handleChange2 = this.handleChange2.bind(this);  
}

**getDerivedStateFromProps**

使用方式：

*//static getDerivedStateFromProps(nextProps, prevState)*  
  
class Example extends React.Component {  
  static getDerivedStateFromProps(props, state) {  
    *//根据 nextProps 和 prevState 计算出预期的状态改变，返回结果会被送给 setState*  
    *// ...*  
  }  
}

新的 getDerivedStateFromProps 是一个静态函数，所以不能在这函数里使用 this，简单来说就是一个纯函数。也表明了 React 团队想通过这种方式防止开发者滥用这个生命周期函数。每当父组件引发当前组件的渲染过程时，getDerivedStateFromProps 会被调用，这样我们有一个机会可以根据新的 props 和当前的 state 来调整新的 state。

这个函数会返回一个对象用来更新当前的 state，如果不需要更新可以返回 null。这个生命周期函数用得比较少，主要用于在重新渲染期间手动对滚动位置进行设置等场景中。该函数会在挂载时，在更新时接收到新的 props，调用了 setState 和 forceUpdate 时被调用。

getDerivedStateFromProps

新的 getDerivedStateFromProps 实际上与 componentDidUpdate 一起取代了以前的 UNSAFE\_componentWillReceiveProps 函数。UNSAFE\_componentWillReceiveProps 也是考虑到因为父组件引发渲染可能要根据 props 更新 state 的需要而设立的。

**~~UNSAVE\_componentWillMount~~**

UNSAFE\_componentWillMount() 在挂载之前被调用。它在 render() 之前调用，因此在此方法中同步调用 setState() 不会触发额外渲染。通常，我们建议使用 constructor() 来初始化 state。避免在此方法中引入任何副作用或订阅。如遇此种情况，请改用 componentDidMount()。

此方法是服务端渲染唯一会调用的生命周期函数。UNSAFE\_componentWillMount() 常用于当支持服务器渲染时，需要同步获取数据的场景。

**render**

这是 React 中最核心的方法，class 组件中唯一必须实现的方法。

当 render 被调用时，它会检查 this.props 和 this.state 的变化并返回以下类型之一：

* 原生的 DOM，如 div
* React 组件
* 数组或 Fragment
* Portals（插槽）
* 字符串和数字，被渲染成文本节点
* Boolean 或 null，不会渲染任何东西

render() 函数应该是一个纯函数，里面只做一件事，就是返回需要渲染的东西，不应该包含其它的业务逻辑，如数据请求，对于这些业务逻辑请移到 componentDidMount 和 componentDidUpdate 中。

**componentDidMount**

componentDidMount() 会在组件挂载后（插入 DOM 树中）立即调用。依赖于 DOM 节点的初始化应该放在这里。如需通过网络请求获取数据，此处是实例化请求的好地方。这个方法是比较适合添加订阅的地方。如果添加了订阅，请不要忘记在 componentWillUnmount() 里取消订阅。

你可以在 componentDidMount() 里直接调用 setState()。它将触发额外渲染，但此渲染会发生在浏览器更新屏幕之前。如此保证了即使在 render() 两次调用的情况下，用户也不会看到中间状态。

请谨慎使用该模式，因为它会导致性能问题。通常，你应该在 constructor() 中初始化 state。如果你的渲染依赖于 DOM 节点的大小或位置，比如实现 modals 和 tooltips 等情况下，你可以使用此方式处理

**更新阶段**

更新阶段是指当组件的 props 发生了改变，或组件内部调用了 setState 或者发生了 forceUpdate，则进行更新。

这个阶段的过程如下：

* ~~UNSAFE\_componentWillReceiveProps~~
* getDerivedStateFromProps
* shouldComponentUpdate
* ~~UNSAFE\_componentWillUpdate~~
* render
* getSnapshotBeforeUpdate
* (React Updates DOM and refs)
* componentDidUpdate

**~~UNSAFE\_componentWillReceiveProps~~**

UNSAFE\_componentWillReceiveProps 是考虑到因为父组件引发渲染可能要根据 props 更新state 的需要而设立的。UNSAFE\_componentWillReceiveProps 会在已挂载的组件接收新的 props 之前被调用。如果你需要更新状态以响应 prop 更改（例如，重置它），你可以比较 this.props 和 nextProps 并在此方法中使用 this.setState() 执行 state 转换。

如果父组件导致组件重新渲染，即使 props 没有更改，也会调用此方法。如果只想处理更改，请确保进行当前值与变更值的比较。在挂载过程中，React 不会针对初始 props 调用 UNSAFE\_componentWillReceiveProps()。组件只会在组件的 props 更新时调用此方法。调用this.setState() 通常不会触发 UNSAFE\_componentWillReceiveProps()。

**getDerivedStateFromProps**

这个方法在挂载阶段已经讲过了，这里不再赘述。记住该函数会在挂载时，在更新时接收到新的 props，调用了 setState 和 forceUpdate 时被调用。它与 componentDidUpdate 一起取代了以前的 UNSAFE\_componentWillReceiveProps 函数。

**shouldComponentUpdate**

shouldComponentUpdate(nextProps, nextState)

它有两个参数，根据此函数的返回值来判断是否进行重新渲染，true 表示重新渲染，false 表示不重新渲染，默认返回 true。注意，首次渲染或者当我们调用forceUpdate 时并不会触发此方法。此方法仅用于性能优化。

因为默认是返回 true，也就是只要接收到新的属性和调用了 setState 都会触发重新的渲染，这会带来一定的性能问题，所以我们需要将 this.props 与 nextProps 以及this.state 与 nextState 进行比较来决定是否返回 false，来减少重新渲染，以优化性能。请注意，返回 false 并不会阻止子组件在 state 更改时重新渲染。

但是官方提倡我们使用内置的 PureComponent 来减少重新渲染的次数，而不是手动编写 shouldComponentUpdate 代码。PureComponent 内部实现了对 props 和 state 进行浅层比较。

如果 shouldComponentUpdate() 返回 false，则不会调用 UNSAFE\_componentWillUpdate()，render() 和 componentDidUpdate()。官方说在后续版本，React 可能会将 shouldComponentUpdate 视为提示而不是严格的指令，并且，当返回 false 时，仍可能导致组件重新渲染。

**~~UNSAFE\_componentWillUpdate~~**

当组件收到新的 props 或 state 时，会在渲染之前调用 UNSAFE\_componentWillUpdate()。使用此作为在更新发生之前执行准备更新的机会。初始渲染不会调用此方法。但是你不能此方法中调用 this.setState()。在 UNSAFE\_componentWillUpdate() 返回之前，你也不应该执行任何其他操作（例如，dispatch Redux 的 action）触发对 React 组件的更新。

通常，此方法可以替换为 componentDidUpdate()。如果你在此方法中读取 DOM 信息（例如，为了保存滚动位置），则可以将此逻辑移至 getSnapshotBeforeUpdate() 中。

**render**

这个方法在挂载阶段已经讲过了，这里不再赘述。

**getSnapshotBeforeUpdate**

getSnapshotBeforeUpdate(prevProps, prevState) {

getSnapshotBeforeUpdate 生命周期方法在 render 之后，在更新之前（如：更新 DOM 之前）被调用。给了一个机会去获取 DOM 信息，计算得到并返回一个 snapshot，这个 snapshot 会作为 componentDidUpdate 的第三个参数传入。如果你不想要返回值，请返回null，不写的话控制台会有警告。

并且，这个方法一定要和 componentDidUpdate 一起使用，否则控制台也会有警告。getSnapshotBeforeUpdate 与 componentDidUpdate 一起，这个新的生命周期涵盖过时的 UNSAFE\_componentWillUpdate 的所有用例。

getSnapshotBeforeUpdate(prevProps, prevState) {  
  console.log('#enter getSnapshotBeforeUpdate');  
  return 'foo';  
}  
  
componentDidUpdate(prevProps, prevState, snapshot) {  
  console.log('#enter componentDidUpdate snapshot = ', snapshot);  
}

上面这段代码可以看出来这个 snapshot 怎么个用法，snapshot 乍一看还以为是组件级别的某个“快照”，其实可以是任何值，到底怎么用完全看开发者自己，getSnapshotBeforeUpdate 把 snapshot 返回，然后 DOM 改变，然后 snapshot 传递给 componentDidUpdate。

官方给了一个例子，用 getSnapshotBeforeUpdate 来处理 scroll，并且说明了通常不需要这个函数，只有在重新渲染过程中手动保留滚动位置等情况下非常有用，所以大部分开发者都用不上，也就不要乱用。

**componentDidUpdate**

componentDidUpdate(prevProps, prevState, snapshot)

componentDidUpdate() 会在更新后会被立即调用。首次渲染不会执行此方法。在这个函数里我们可以操作 DOM，和发起服务器请求，还可以 setState，但是注意一定要用 if 语句控制，否则会导致无限循环。

componentDidUpdate(prevProps) {  
  *// 典型用法（不要忘记比较 props）：*  
  if (this.props.userID !== prevProps.userID) {  
    this.fetchData(this.props.userID);  
  }  
}

如果组件实现了 getSnapshotBeforeUpdate() 生命周期，则它的返回值将作为 componentDidUpdate() 的第三个参数 snapshot 参数传递。否则此参数将为 undefined。

**卸载阶段**

卸载阶段，这个阶段的生命周期函数只有一个：

**componentWillUnmount**

componentWillUnmount() 会在组件卸载及销毁之前直接调用。我们可以在此方法中执行必要的清理操作，例如，清除 timer，取消网络请求或清除在 componentDidMount() 中创建的订阅等。注意不要在这个函数里调用 setState()，因为组件不会重新渲染了。

**其他不常用的生命周期函数**

还有两个很不常用的生命周期函数，在这也列一下。

详细使用示例请见：**React 官方文档[3]**

**static getDerivedStateFromError()**

static getDerivedStateFromError(error)

此生命周期会在后代组件抛出错误后被调用。它将抛出的错误作为参数，并返回一个值以更新state。getDerivedStateFromError() 会在渲染阶段调用，因此不允许出现副作用。如遇此类情况，请改用 componentDidCatch()。

**componentDidCatch()**

componentDidCatch(error, info)

此生命周期在后代组件抛出错误后被调用。它接收两个参数：

1. error —— 抛出的错误。
2. info —— 带有 componentStack key 的对象，其中包含有关组件引发错误的栈信息。

componentDidCatch() 会在“提交”阶段被调用，因此允许执行副作用。它应该用于记录错误之类的情况：

如果发生错误，你可以通过调用 setState 使用 componentDidCatch() 渲染降级 UI，但在未来的版本中将不推荐这样做。可以使用静态 getDerivedStateFromError() 来处理降级渲染。

**参考资料**

* **React.Component — React Docs[4]**
* **异步渲染之更新 — React Docs[5]**
* **Update on Async Rendering — React Docs[6]**
* **React v16.3 之后的组件生命周期函数[7]**
* **谈谈 React 新的生命周期钩子[8]**
* **我对 React v16.4 生命周期的理解[9]**

**结语**

有人会说，现在都 Hooks 一把梭了，你总结整合这些内容有啥用。其实学习这些内容，能够帮助你加深对 React 的理解，深入领会 React 的思想。并且，目前 Class component 与 Hooks 是并存的，虽然新项目一般都直接用 Hooks，但是老项目中难免会遇到 Class component，所以还是要学会的。

**参考资料**

[1]

异步渲染之更新 — React Docs: https://zh-hans.reactjs.org/blog/2018/03/27/update-on-async-rendering.html

[2]

Update on Async Rendering — React Docs: https://reactjs.org/blog/2018/03/27/update-on-async-rendering.html#initializing-state

[3]

React 官方文档: https://zh-hans.reactjs.org/docs/react-component.html#static-getderivedstatefromerror

[4]

React.Component — React Docs: https://zh-hans.reactjs.org/docs/react-component.html

[5]

异步渲染之更新 — React Docs: https://zh-hans.reactjs.org/blog/2018/03/27/update-on-async-rendering.html

[6]

Update on Async Rendering — React Docs: https://reactjs.org/blog/2018/03/27/update-on-async-rendering.html#initializing-state

[7]

React v16.3 之后的组件生命周期函数: https://zhuanlan.zhihu.com/p/38030418

[8]

谈谈 React 新的生命周期钩子: https://zhuanlan.zhihu.com/p/42413419

[9]

我对 React v16.4 生命周期的理解: https://juejin.im/post/5b6f1800f265da282d45a79a

**看完三件事**

如果你觉得本文对你有帮助，我想请你帮个忙：

1. 转发本文或者点个「**在看**」，让更多的人也能看到这篇内容；
2. 关注公众号「**技术漫谈**」，订阅更多精彩内容，获取更多学习资料；
3. 可以在本公众号后台回复「**加群**」，加入技术群，与更多读者交流。

**长按上方二维码关注公众号「技术漫谈」，订阅更多精彩内容。**

**你“在看”我吗？**

#源码解读##前端##React#

阅读 242

赞2在看7