react100道面试题-2（https://github.com/minsion）

面试题 1. 简述componentWillReceiveProps的调用时机?

推荐指数：★★ 试题难度：高难 试题类型：原理题

试题回答参考思路：

componentWillReceiveProps这个生命周期在16.3之后已经被废弃，在组件接受到一个新的props时被调用，当组件初始化render的时候不会被调用

面试题 2. 执行两次setState的时候会render几次? 会不会立即触发?

推荐指数：★★★★ 试题难度：初级 试题类型：原理题

试题回答参考思路：

只执行一次，不会立即触发，因为react中有批处理机制，React会把setState的调用合并为一个来执行，也就是说，当执行setState的时候，state中的数据并不会马上更新，会按照先进先出，按顺序进行执行，但是在 Ajax、setTimeout 等异步方法中，每 setState 一次，就会 re-render 一次

面试题 3. 简述React.memo()和React.PureComponent组件异同 ?

推荐指数：★★ 试题难度：高难 试题类型：原理题

试题回答参考思路：

异：React.memo()是函数组件，React.PureComponent是类组件。
同：都是对接收的props参数进行浅比较，解决组件在运行时的效率问题，优化组件的重渲染行为。
useMemo()是定义一段函数逻辑是否重复执行。
若第二个参数为空数组，则只会在渲染组件时执行一次，传入的属性值的更新也不会有作用。 所以useMemo()的第二个参数，数组中需要传入依赖的参数。

面试题 4. React 什么是 Reselect 以及它是如何工作的 ?

推荐指数：★★★ 试题难度：中级 试题类型：原理题

试题回答参考思路：

*Reselect*是一个**选择器库**（用于 Redux ），它使用*memoization*概念。它最初编写用于计算类似 Redux 的应用程序状态的派生数据，但它不能绑定到任何体系结构或库。

Reselect 保留最后一次调用的最后输入/输出的副本，并仅在其中一个输入发生更改时重新

计算结果。如果连续两次提供相同的输入，则 Reselect 将返回缓存的输出。它的 memoization 和缓存是完全可定制的

面试题 5. 在React中如何防范XSS攻击?

推荐指数: ★★★ 试题难度: 高难 试题类型: 原理题

试题回答参考思路:

dangerouslySetInnerHTML
dangerouslySetInnerHTML 是 React 解析含有 HTML 标记内容的一种方式，也是原生 DOM 元素 innerHTML 的替代方案

面试题 6. 简述点(...)在 React 的作用 ?

推荐指数: ★★★★★ 试题难度: 高难 试题类型: 原理题

试题回答参考思路:

扩展运算符或者叫展开操作符，对于创建具有现有对象的大多数属性的新对象非常方便，在更新state时经常这么用

面试题 7. 简述什么是prop drilling，如何避免?

推荐指数: ★★ 试题难度: 高难 试题类型: 原理题

试题回答参考思路:

在构建 React 应用程序时，在多层嵌套组件来使用另一个嵌套组件提供的数据。最简单的方法是将一个 prop 从每个组件一层层的传递下去，从源组件传递到深层嵌套组件，这叫做 prop drilling。

prop drilling的主要缺点是原本不需要数据的组件变得不必要地复杂，并且难以维护。

为了避免prop drilling，一种常用的方法是使用React Context。通过定义提供数据的Provider组件，并允许嵌套的组件通过Consumer组件或useContext Hook 使用上下文数据

面试题 8. 简述什么是 React Fiber?

推荐指数: ★★ 试题难度: 中级 试题类型: 原理题

试题回答参考思路:

Fiber 是 React 16 中新的协调引擎或重新实现核心算法。它的主要目标是支持虚拟DOM的增量渲染。React Fiber 的目标是提高其在动画、布局、手势、暂停、中止或重用等方面的适用性，并为不同类型的更新分配优先级，以及新的并发原语。

React Fiber 的目标是增强其在动画、布局和手势等领域的适用性。它的主要特性是增量渲染:能够将渲染工作分割成块，并将其分散到多个帧中

面试题 9. 如何在 React 的 Props上应用验证?

推荐指数: ★★★★ 试题难度: 中级 试题类型: 原理题

试题回答参考思路:

当应用程序在开发模式下运行时，React 将自动检查咱们在组件上设置的所有 props，以确保它们具有正确的数据类型。对于不正确的类型，开发模式下会在控制台中生成警告消息，而在生产模式中由于性能影响而禁用它。强制的 props 用 isRequired定义的。

下面是一组预定义的 prop 类型:

React.PropTypes.string
React.PropTypes.number
React.PropTypes.func
React.PropTypes.node
React.PropTypes.bool

面试题 10. React 中使用构造函数和 getInitialState 有什么区别?

推荐指数: ★★★ 试题难度: 初级 试题类型: 原理题

试题回答参考思路:

构造函数和getInitialState之间的区别就是ES6和ES5本身的区别。在使用ES6类时，应该在构造函数中初始化state，并在使用React.createClass时定义getInitialState方法

面试题 11. 解释Hooks会取代 render props 和高阶组件吗?

推荐指数: ★★★ 试题难度: 高难 试题类型: 原理题

试题回答参考思路:

render props和高阶组件仅渲染一个子组件。React团队认为，Hooks 是服务此用例的更简单方法。

这两种模式仍然有一席之地(例如，一个虚拟的 scroller 组件可能有一个 renderItem prop，或者一个可视化的容器组件可能有它自己的 DOM 结构)。但在大多数情况下，Hooks 就足够了，可以帮助减少树中的嵌套

面试题 12. 如何避免React 组件的重新渲染?

推荐指数: ★★★★★ 试题难度: 中级 试题类型: 原理题

试题回答参考思路：

React 中最常见的问题之一是组件不必要地重新渲染。React 提供了两个方法，在这些情况下非常有用：

React.memo():这可以防止不必要地重新渲染函数组件
PureComponent:这可以防止不必要地重新渲染类组件
这两种方法都依赖于对传递给组件的props的浅比较，如果 props 没有改变，那么组件将不会重新渲染。虽然这两种工具都非常有用，但是浅比较会带来额外的性能损失，因此如果使用不当，这两种方法都会对性能产生负面影响。

通过使用 React Profiler，可以在使用这些方法前后对性能进行测量，从而确保通过进行给定的更改来实际改进性能。

面试题 13. 请简述当调用setState时，React render 是如何工作的?

推荐指数：★★★★ 试题难度： 初级 试题类型： 原理题

试题回答参考思路：

1 虚拟 DOM 渲染:当render方法被调用时，它返回一个新的组件的虚拟 DOM 结构。当调用setState()时，render会被再次调用，因为默认情况下shouldComponentUpdate总是返回true，所以默认情况下 React 是没有优化的。
2 原生 DOM 渲染:React 只会在虚拟DOM中修改真实DOM节点，而且修改的次数非常少——这是很棒的React特性，它优化了真实DOM的变化，使React变得更快

面试题 14. 解释如何避免在React重新绑定实例?

推荐指数：★★★★★ 试题难度： 中级 试题类型： 原理题

试题回答参考思路：

1.将事件处理程序定义为内联箭头函数

```
class SubmitButton extends React.Component {
constructor(props) {
super(props);
this.state = {
isFormSubmitted: false
}
;
}
render() {
return (
```

```
() => {
this.setState( {
isFormSubmitted: true
}
);
}
}
>Submit
)
}
}
```

2.使用箭头函数来定义方法：

```
class SubmitButton extends React.Component {
state = {
isFormSubmitted: false
}
handleSubmit = () => {
this.setState( {
isFormSubmitted: true
}
);
}
render() {
return (
this.handleSubmit
}
>Submit
)
}
}
```

3.使用带有 Hooks 的函数组件

```
const SubmitButton = () => {
const [isFormSubmitted, setIsFormSubmitted] = useState(false);
return (
() => {
setIsFormSubmitted(true);
}
}
>Submit
)
}
;
```

面试题 15. 简述React 中的 useState() 是什么?

推荐指数：★★★★ 试题难度：初级 试题类型：原理题

试题回答参考思路：

useState 是一个内置的 React Hook。useState(0) 返回一个元组，其中第一个参数count是计数器的当前状态，setCounter 提供更新计数器状态的方法。

咱们可以在任何地方使用setCounter方法更新计数状态-在这种情况下，咱们在setCount函数内部使用它可以做更多的事情，使用 Hooks，能够使咱们的代码保持更多功能，还可以避免过多使用基于类的组件
定义state的数据，参数是初始化的数据，返回值两个值1. 初始化值，2. 修改的方法
useState中修改的方法异步
借助于useEffect 进行数据的监听
可以自己定义Hooks的方法，方法内部可以把逻辑返回

面试题 16. Component, Element, Instance 之间有什么区别和联系?

推荐指数： ★★★★ 试题难度： 中级 试题类型： 原理题

试题回答参考思路：

元素：一个元素element是一个普通对象(plain object)，描述了对于一个DOM节点或者其他组件component，你想让它在屏幕上呈现成什么样子。元素element可以在它的属性props中包含其他元素(译注:用于形成元素树)。创建一个React元素element成本很低。元素element创建之后是不可变的。

组件：一个组件component可以通过多种方式声明。可以是带有一个render()方法的类，简单点也可以定义为一个函数。这两种情况下，它都把属性props作为输入，把返回的一棵元素树作为输出。

实例：一个实例instance是你在所写的组件类component class中使用关键字this所指向的东西(译注:组件实例)。它用来存储本地状态和响应生命周期事件很有用。

函数式组件(Functional component)根本没有实例instance。类组件(Class component)有实例instance，但是永远也不需要直接创建一个组件的实例，因为React帮我们做了这些。

面试题 17. 简述React.createClass和extends Component的区别有哪些?

推荐指数： ★★★ 试题难度： 高难 试题类型： 原理题

试题回答参考思路：

React.createClass和extends Component的bai区别主要在于：
（1）语法区别
createClass本质上是一个工厂函数，extends的方式更加接近最新的ES6规范的class写法。两种方式在语法上的差别主要体现在方法的定义和静态属性的声明上。
createClass方式的方法定义使用逗号，隔开，因为creatClass本质上是一个函数，传递给它的是一个Object；而class的方式定义方法时务必谨记不要使用逗号隔开，这是ES6 class的语法规范。
（2）propType 和 getDefaultProps
React.createClass：通过proTypes对象和getDefaultProps()方法来设置和获取props.
React.Component：通过设置两个属性propTypes和defaultProps
（3）状态的区别

React.createClass：通过getInitialState()方法返回一个包含初始值的对象
React.Component：通过constructor设置初始状态
（4）this区别
React.createClass：会正确绑定this
React.Component：由于使用了 ES6，这里会有些微不同，属性并不会自动绑定到 React 类的实例上。
（5）Mixins
React.createClass：使用 React.createClass 的话，可以在创建组件时添加一个叫做 mixins 的属性，并将可供混合的类的集合以数组的形式赋给 mixins。
如果使用 ES6 的方式来创建组件，那么 React mixins 的特性将不能被使用了。

面试题 18. 哪些方法会触发 React 重新渲染？重新渲染 render 会做些什么？

推荐指数：★★★★★ 试题难度：中级 试题类型：原理题

试题回答参考思路：

（1）哪些方法会触发 react 重新渲染？

setState（）方法被调用
setState 是 React 中最常用的命令，通常情况下，执行 setState 会触发 render。但是这里有个点值得关注，执行 setState 的时候不一定会重新渲染。当 setState 传入 null 时，并不会触发 render。

```
class App extends React.Component {
state = {
a: 1
};

render() {
console.log("render");
return (


{this.state.a}


onClick={() => {
this.setState({ a: 1 }); // 这里并没有改变 a 的值
}}
>
Click me

this.setState(null)}>setState null


);
}
```

}
父组件重新渲染

只要父组件重新渲染了，即使传入子组件的 props 未发生变化，那么子组件也会重新渲染，进而触发 render

（2）重新渲染 render 会做些什么？

会对新旧 VNode 进行对比，也就是我们所说的Diff算法。

对新旧两棵树进行一个深度优先遍历，这样每一个节点都会一个标记，在到深度遍历的时候，每遍历到一和个节点，就把该节点和新的节点树进行对比，如果有差异就放到一个对象里面

遍历差异对象，根据差异的类型，根据对应对规则更新VNode

React 的处理 render 的基本思维模式是每次一有变动就会去重新渲染整个应用。在 Virtual DOM 没有出现之前，最简单的方法就是直接调用 innerHTML。Virtual DOM厉害的地方并不是说它比直接操作 DOM 快，而是说不管数据怎么变，都会尽量以最小的代价去更新 DOM。React 将 render 函数返回的虚拟 DOM 树与老的进行比较，从而确定 DOM 要不要更新、怎么更新。当 DOM 树很大时，遍历两棵树进行各种比对还是相当耗性能的，特别是在顶层 setState 一个微小的修改，默认会去遍历整棵树。尽管 React 使用高度优化的 Diff 算法，但是这个过程仍然会损耗性能.

面试题 19. React如何判断什么时候重新渲染组件?

推荐指数：★★★ 试题难度： 中级 试题类型： 原理题

试题回答参考思路：

组件状态的改变可以因为props的改变，或者直接通过setState方法改变。组件获得新的状态，然后React决定是否应该重新渲染组件。只要组件的state发生变化，React就会对组件进行重新渲染。这是因为React中的shouldComponentUpdate方法默认返回true，这就是导致每次更新都重新渲染的原因。

当React将要渲染组件时会执行shouldComponentUpdate方法来看它是否返回true（组件应该更新，也就是重新渲染）。所以需要重写shouldComponentUpdate方法让它根据情况返回true或者false来告诉React什么时候重新渲染什么时候跳过重新渲染

面试题 20. 简述对React中Fragment的理解，它的使用场景是什么?

推荐指数：★★★★★ 试题难度： 高难 试题类型： 原理题

试题回答参考思路：

在React中，组件返回的元素只能有一个根元素。为了不添加多余的DOM节点，我们可以使用Fragment标签来包裹所有的元素，Fragment标签不会渲染出任何元素。React官方对Fragment的解释：

React 中的一个常见模式是一个组件返回多个元素。Fragments 允许你将子列表分组，而无需向 DOM 添加额外节点。

```
import React, { Component, Fragment } from 'react'

// 一般形式
render() {
return (




);
}
// 也可以写成以下形式
render() {
return (
<>




);
}
```

面试题 21. React中可以在render访问refs吗?

推荐指数: ★★★ 试题难度: 初级 试题类型: 原理题

试题回答参考思路:

```
<>
{this.state.title}
{
this.spanRef.current ? '有值' : '无值'
}
```

不可以，render 阶段 DOM 还没有生成，无法获取 DOM。DOM 的获取需要在 pre-commit 阶段和 commit 阶段

面试题 22. 简述React的插槽(Portals)的理解?

推荐指数: ★★★★ 试题难度: 中级 试题类型: 原理题

试题回答参考思路:

插槽：将一个React元素渲染到指定的Dom容器中
ReactDOM.createPortal(React元素，真实的DOM容器)，该函数返回一个React元素
第一个参数（child）是任何可渲染的 React 子元素，例如一个元素，字符串或fragment。
第二个参数（container）是一个 DOM 元素。

```
import React, { Component } from "react";
import ReactDOM from "react-dom";

export default class Portals extends Component {
render() {
return (
onClick={() => {
console.log("rooter click");
}}
>

我想出现在root中




);
}
}

function Test() {
return ReactDOM.createPortal(
,
//
```

我想出现在container中

```
,
document.getElementById("container")
);
}

function ChildA() {
return

我是childA

;
}
```

面试题 23. 简述对React-Intl 的理解，它的工作原理?

推荐指数：★★★★ 试题难度：高难 试题类型：原理题

试题回答参考思路：

React-intl是雅虎的语言国际化开源项目FormatJS的一部分，通过其提供的组件和API可以与ReactJS绑定。

React-intl提供了两种使用方法，一种是引用React组件，另一种是直接调取API，官方更加推荐在React项目中使用前者，只有在无法使用React组件的地方，才应该调用框架提供的API。它提供了一系列的React组件，包括数字格式化、字符串格式化、日期格式化等。

在React-intl中，可以配置不同的语言包，他的工作原理就是根据需要，在语言包之间进行切换

面试题 24. React 并发模式是如何执行的?

推荐指数：★★★★★ 试题难度：高难 试题类型：原理题

试题回答参考思路：

React 中的并发，并不是指同一时刻同时在做多件事情。因为 js 本身就是单线程的（同一时间只能执行一件事情），而且还要跟 UI 渲染竞争主线程。若一个很耗时的任务占据了线程，那么后续的执行内容都会被阻塞。为了避免这种情况，React 就利用 fiber 结构和时间切片的机制，将一个大任务分解成多个小任务，然后按照任务的优先级和线程的占用情况，对任务进行调度。

对于每个更新，为其分配一个优先级 lane，用于区分其紧急程度。
通过 Fiber 结构将不紧急的更新拆分成多段更新，并通过宏任务的方式将其合理分配到浏览器的帧当中。这样就能使得紧急任务能够插入进来。
高优先级的更新会打断低优先级的更新，等高优先级更新完成后，再开始低优先级更新

面试题 25. React setState 调用之后发生了什么？是同步还是异步?

推荐指数：★★★★ 试题难度：中级 试题类型：原理题

试题回答参考思路：

（1）React中setState后发生了什么
在代码中调用setState函数之后，React 会将传入的参数对象与组件当前的状态合并，然后触发调和过程(Reconciliation)。经过调和过程，React 会以相对高效的方式根据新的状态构建 React 元素树并且着手重新渲染整个UI界面。
在 React 得到元素树之后，React 会自动计算出新的树与老树的节点差异，然后根据差异对界面进行最小化重渲染。在差异计算算法中，React 能够相对精确地知道哪些位置发生了改变以及应该如何改变，这就保证了按需更新，而不是全部重新渲染。
如果在短时间内频繁setState。React会将state的改变压入栈中，在合适的时机，批量更新state和视图，达到提高性能的效果。
（2）setState 是同步还是异步的
假如所有setState是同步的，意味着每执行一次setState时（有可能一个同步代码中，多次setState），都重新vnode diff + dom修改，这对性能来说是极为不好的。如果是异步，

则可以把一个同步代码中的多个setState合并成一次组件更新。所以默认是异步的，但是在一些情况下是同步的。
setState 并不是单纯同步/异步的，它的表现会因调用场景的不同而不同。在源码中，通过 isBatchingUpdates 来判断setState 是先存进 state 队列还是直接更新，如果值为 true 则执行异步操作，为 false 则直接更新。
异步： 在 React 可以控制的地方，就为 true，比如在 React 生命周期事件和合成事件中，都会走合并操作，延迟更新的策略。
同步： 在 React 无法控制的地方，比如原生事件，具体就是在 addEventListener 、setTimeout、setInterval 等事件中，就只能同步更新。
一般认为，做异步设计是为了性能优化、减少渲染次数：
setState设计为异步，可以显著的提升性能。如果每次调用 setState都进行一次更新，那么意味着render函数会被频繁调用，界面重新渲染，这样效率是很低的；最好的办法应该是获取到多个更新，之后进行批量更新；
如果同步更新了state，但是还没有执行render函数，那么state和props不能保持同步。
state和props不能保持一致性，会在开发中产生很多的问题；

面试题 26. 简述super()和super(props)有什么区别?

推荐指数： ★★★★ 试题难度： 中级 试题类型： 原理题

试题回答参考思路：

在ES6中，通过extends关键字实现类的继承,super关键字实现调用父类，super代替的是父类的构建函数，使用super(xx)相当于调用sup.prototype.constructor.call(this.xx)，如果在子类中不使用super关键字，则会引发报错
super()就是将父类中的this对象继承给子类的，没有super()子类就得不到this对象

在React中，类组件是基于es6的规范实现的，继承React.Component，因此如果用到constructor就必须写super()才初始化this，在调用super()的时候，我们一般都需要传入props作为参数，如果不传进去，React内部也会将其定义在组件实例中，所以无论有没有constructor，在render中this.props都是可以使用的，这是React自动附带的，但是也不建议使用super()代替super(props)，因为在React会在类组件构造函数生成实例后再给this.props赋值，所以在不传递props在super的情况下，调用this.props为undefined,而传入props的则都能正常访问，确保了 this.props 在构造函数执行完毕之前已被赋值，更符合逻辑

总结

在React中，类组件基于ES6，所以在constructor中必须使用super
在调用super过程，无论是否传入props，React内部都会将porps赋值给组件实例porps属性中
如果只调用了super()，那么this.props在super()和构造函数结束之间仍是undefined

面试题 27. 简述React中组件间过渡动画如何实现?

推荐指数： ★★★ 试题难度： 中级 试题类型： 原理题

试题回答参考思路：

在react中，react-transition-group是一种很好的解决方案，其为元素添加enter，enter-active，exit，exit-active这一系列勾子，可以帮助我们方便的实现组件的入场和离场动画

其主要提供了三个主要的组件：

CSSTransition：在前端开发中，结合 CSS 来完成过渡动画效果
SwitchTransition：两个组件显示和隐藏切换时，使用该组件
TransitionGroup：将多个动画组件包裹在其中，一般用于列表中元素的动画

面试题 28. 简述如何Redux 中的异步请求 ？

推荐指数：★★★★★ 试题难度： 中级 试题类型： 原理题

试题回答参考思路：

在 Redux 的应用中，我们经常需要进行异步请求，如获取数据、发送请求等。然而，Redux 原生并不支持异步请求，因为 Redux 中的数据流是单向的，由 View、Action、Reducer 三个部分组成。Action 触发 Reducer 更新 State，从而触发 View 重新渲染。如果 Action 中包含异步请求，就需要将这个请求和处理请求的数据的过程放在远程服务器上。因此，Redux 本身并不处理异步请求，它需要依赖其他的库来支持异步操作。

常见的处理异步请求的库是 Redux-thunk 和 Redux-saga。其中，Redux-thunk 是 Redux 官方推荐的一个异步操作的中间件。它允许我们在 Action 中编写异步代码，使得 Action 可以返回一个函数而不是一个对象。在函数中，我们可以发起异步请求、处理异步请求的数据，然后使用 Dispatch 将数据传递给 Reducer 更新 State。而 Redux-saga 则是一个更加强大的异步操作库，它使用了 ES6 中的 Generator 函数来实现异步操作。相对于 Redux-thunk，Redux-saga 提供了更多的功能，比如取消异步操作、自动重试等

面试题 29. React.forwardRef是什么？它有什么作用?

推荐指数：★★★ 试题难度： 初级 试题类型： 原理题

试题回答参考思路：

React.forwardRef 会创建一个React组件，这个组件能够将其接受的 ref 属性转发到其组件树下的另一个组件中。这种技术并不常见，但在以下两种场景中特别有用：
1 转发 refs 到 DOM 组件
2 在高阶组件中转发 refs

面试题 30. 简述React的状态提升是什么？使用场景有哪些?

推荐指数：★★★★★ 试题难度： 中级 试题类型： 原理题

试题回答参考思路：

React的状态提升就是用户对子组件操作，子组件不改变自己的状态，通过自己的props把这个操作改变的数据传递给父组件，改变父组件的状态，从而改变受父组件控制的所有子组件的状态，这也是React单项数据流的特性决定的。官方的原话是：共享 state(状态) 是通过将其移动到需要它的组件的最接近的共同祖先组件来实现的。 这被称为“状态提升(Lifting State Up)”。

概括来说就是将多个组件需要共享的状态提升到它们最近的父组件上，在父组件上改变这个状态然后通过props分发给子组件。

一个简单的例子，父组件中有两个input子组件，如果想在第一个输入框输入数据，来改变第二个输入框的值，这就需要用到状态提升。

```
class Father extends React.Component {
constructor(props) {
super(props)
this.state = {
Value1: '',
Value2: ''
}
}
value1Change(aa) {
this.setState({
Value1: aa
})
}
value2Change(bb) {
this.setState({
Value2: bb
})
}
render() {
return (




)
}
}
class Child1 extends React.Component {
constructor(props) {
super(props)
}
changeValue(e) {
this.props.onvalue1Change(e.target.value)
}
render() {
```

```
return (
{this.props.Value1}
)
}
}
class Child2 extends React.Component {
constructor(props) {
super(props)
}
render() {
return (
{this.props.value2}
)
}
}

ReactDOM.render(
,
document.getElementById('root')
)
```

面试题 31. React 中的高阶组件运用了什么设计模式?

推荐指数: ★★★★★ 试题难度: 高难 试题类型: 原理题

试题回答参考思路:

使用了装饰模式，高阶组件的运用:

```
function withWindowWidth(BaseComponent) {
class DerivedClass extends React.Component {
state = {
windowWidth: window.innerWidth,
}
onResize = () => {
this.setState({
windowWidth: window.innerWidth,
})
}
componentDidMount() {
window.addEventListener('resize', this.onResize)
}
componentWillUnmount() {
window.removeEventListener('resize', this.onResize);
}
render() {
return
}
```

```
}
return DerivedClass;
}
const MyComponent = (props) => {
return

Window width is: {props.windowWidth}

};
export default withWindowWidth(MyComponent);
```

装饰模式的特点是不需要改变 被装饰对象 本身，而只是在外面套一个外壳接口。JavaScript 目前已经有了原生装饰器的提案，其用法如下：

```
@testable
class MyTestableClass {
}
```

面试题 32. React中constructor和getInitialState的区别?

推荐指数：★★★★ 试题难度：高难 试题类型：原理题 ▸

试题回答参考思路：

两者都是用来初始化state的。前者是ES6中的语法，后者是ES5中的语法，新版本的React中已经废弃了该方法。

getInitialState是ES5中的方法，如果使用createClass方法创建一个Component组件，可以自动调用它的getInitialState方法来获取初始化的State对象，

```
var APP = React.creatClass ({
getInitialState() {
return {
userName: 'hi',
userId: 0
};
  }
})
```

React在ES6的实现中去掉了getInitialState这个hook函数，规定state在constructor中实现，如下：

```
Class App extends React.Component{
constructor(props){
super(props);
this.state={};
```

```
}
}
```

面试题 33. React 如何实现强制刷新?

推荐指数: ★★★ 试题难度: 中级 试题类型: 原理题

试题回答参考思路:

component.forceUpdate() 一个不常用的生命周期方法，它的作用就是强制刷新

官网解释如下:

默认情况下，当组件的 state 或 props 发生变化时，组件将重新渲染。如果 render() 方法依赖于其他数据，则可以调用 forceUpdate() 强制让组件重新渲染。
调用 forceUpdate() 将致使组件调用 render() 方法，此操作会跳过该组件的 shouldComponentUpdate()。但其子组件会触发正常的生命周期方法，包括 shouldComponentUpdate() 方法。如果标记发生变化，React 仍将只更新 DOM。
通常你应该避免使用 forceUpdate()，尽量在 render() 中使用 this.props 和 this.state。
shouldComponentUpdate 在初始化 和 forceUpdate 不会执行

面试题 34. 简述React 之 高低版本区别 ?

推荐指数: ★★★★ 试题难度: 中级 试题类型: 原理题

试题回答参考思路:

1、引入文件不同:
高版本: react.production.js、 react-dom.production.min.js、browser.min.js
低版本: react.min.js、react-dom.min.js、browser.min.js
2、创建组件方式不同:
高版本: 通过 类 的继承—class xx extends React.Component {
}
低版本: React.createClass {
}
3、生命周期所用钩子函数不同:
高版本（3个）: componentWillMount、render、componentDidMount
低版本（5个）: getDefaultProps、getInitialState、componentWillMount、render、componentDidMount
4、属性之间传值:
高版本: 用组件defaultProps构造器传值
低版本: 用钩子函数getDefaultProps传值
5、状态state不同:
高版本: 在constructor中一用this.state= {
}
初始状态，调用this.setState()需要在constructor中通过bind绑定this指向

低版本：用getInitialState()初始状态，用this.setState()更新组件的状态并在其内bind绑定this指向

面试题 35. React setState 笔试题，下面的代码输出什么？

```
class Example extends React.Component {
constructor() {
super()
this.state = {
val: 0
}
}
componentDidMount() {
this.setState({ val: this.state.val + 1 })
console.log(this.state.val)
// 第 1 次 log
this.setState({ val: this.state.val + 1 })
console.log(this.state.val)
// 第 2 次 log
setTimeout(() => {
this.setState({ val: this.state.val + 1 })
console.log(this.state.val)
// 第 3 次 log
this.setState({ val: this.state.val + 1 })
console.log(this.state.val)
// 第 4 次 log
}, 0)
}
render() {
return null
}
}
```

推荐指数：★★★★★ 试题难度：中级 试题类型：编程题

试题回答参考思路：

0，0，1，2

面试题 36. 简述React触发多次setstate，那么render会执行几次？

推荐指数：★★★★★ 试题难度：初级 试题类型：原理题

试题回答参考思路：

多次setState会合并为一次render，因为setState并不会立即改变state的值，而是将其放到一个任务队列里，最终将多个setState合并，一次性更新页面。所以我们可以在代码里多次调用setState，每次只需要关注当前修改的字段即可

面试题 37. 简述原生事件和React事件的区别？

推荐指数：★★★★ 试题难度：初级 试题类型：原理题

试题回答参考思路：

React 事件使用驼峰命名，而不是全部小写。
通过 JSX，你传递一个函数作为事件处理程序，而不是一个字符串。
在 React 中你不能通过返回 false 来阻止默认行为。必须明确调用 preventDefault 。

面试题 38. React 高阶组件、Render props、hooks 有什么区别，为什么要 不断迭代？

推荐指数：★★★★ 试题难度：高难 试题类型：原理题

试题回答参考思路：

这三者是目前react解决代码复用的主要方式：
高阶组件（HOC）是 React 中用于复用组件逻辑的一种高级技巧。HOC 自身不是 React API 的
一部分，它是一种基于 React 的组合特性而形成的设计模式。具体而言，高阶组件是参数为组件，
返回值为新组件的函数。
render props是指一种在 React 组件之间使用一个值为函数的 prop 共享代码的简单技术，更
具体的说，render prop 是一个用于告知组件需要渲染什么内容的函数 prop。
通常，render props 和高阶组件只渲染一个子节点。让 Hook 来服务这个使用场景更加简单。这
两种模式仍有用武之地，（例如，一个虚拟滚动条组件或许会有一个 renderItem 属性，或是一个
可见的容器组件或许会有它自己的 DOM 结构）。但在大部分场景下，Hook 足够了，并且能够帮助
减少嵌套。
（1）HOC 官方解释：
高阶组件（HOC）是 React 中用于复用组件逻辑的一种高级技巧。HOC 自身不是 React API
的一部分，它是一种基于 React 的组合特性而形成的设计模式。
简言之，HOC是一种组件的设计模式，HOC接受一个组件和额外的参数（如果需要），返回一个新的组
件。HOC 是纯函数，没有副作用。
HOC的优缺点：

```
// hoc的定义
function withSubscription(WrappedComponent, selectData) {
return class extends React.Component {
constructor(props) {
super(props);
this.state = {
data: selectData(DataSource, props)
};
}
// 一些通用的逻辑处理
render() {
```

```
// ... 并使用新数据渲染被包装的组件!
return ;
}
};
// 使用
const BlogPostWithSubscription = withSubscription(BlogPost,
(DataSource, props) => DataSource.getBlogPost(props.id));
```

复制代码

优点：逻辑服用、不影响被包裹组件的内部逻辑。

缺点：hoc传递给被包裹组件的props容易和被包裹后的组件重名，进而被覆盖

（2）Render props 官方解释：

"render prop"是指一种在 React 组件之间使用一个值为函数的 prop 共享代码的简单技术具有render prop 的组件接受一个返回React元素的函数，将render的渲染逻辑注入到组件内部。在
这里，"render"的命名可以是任何其他有效的标识符。

由此可以看到，render props的优缺点也很明显：

优点：数据共享、代码复用，将组件内的state作为props传递给调用者，将渲染逻辑交给调用者。

缺点：无法在 return 语句外访问数据、嵌套写法不够优雅

（3）Hooks 官方解释：

```
// DataProvider组件内部的渲染逻辑如下
class DataProvider extends React.Components {
state = {
name: 'Tom'
}
render() {
return (



共享数据组件自己内部的渲染逻辑



{ this.props.render(this.state) }

);
}
}
// 调用方式
(
# Hello {data.name}


)}/>
```

复制代码

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。通过自定义hook，可以复用代码逻辑。

以上可以看出，hook解决了hoc的prop覆盖的问题，同时使用的方式解决了render props

的嵌套地狱
的问题。hook的优点如下：
使用直观；
解决hoc的prop 重名问题；
解决render props 因共享数据 而出现嵌套地狱的问题；
能在return之外使用数据的问题。
需要注意的是：hook只能在组件顶层使用，不可在分支语句中使用。
总结：：Hoc、render props和hook都是为了解决代码复用的问题，但是hoc和render props都有特
定的使用场景和明显的缺点。hook是react16.8更新的新的API，让组件逻辑复用更简洁明了，同时也
解决了hoc和render props的一些缺点

面试题 39. 对React-Fiber的理解，它解决了什么问题?

推荐指数： ★★★ 试题难度： 中级 试题类型： 原理题

试题回答参考思路：

React V15 在渲染时，会递归比对 VirtualDOM 树，找出需要变动的节点，然后同步更新它们， 一
气呵成。这个过程期间， React 会占据浏览器资源，这会导致用户触发的事件得不到响应，并且会导致
掉帧，导致用户感觉到卡顿。

```
// 自定义一个获取订阅数据的hook
function useSubscription() {
const data = DataSource.getComments();
return [data];
}
//
function CommentList(props) {
const {data} = props;
const [subData] = useSubscription();
...
}
// 使用
```

复制代码
为了给用户制造一种应用很快的“假象”，不能让一个任务长期霸占着资源。 可以将浏览器的渲染、布
局、绘制、资源加载(例如 HTML 解析)、事件响应、脚本执行视作操作系统的“进程”，需要通过某些调
度策略合理地分配 CPU 资源，从而提高浏览器的用户响应速率，同时兼顾任务执行效率。
所以 React 通过Fiber 架构，让这个执行过程变成可被中断。“适时”地让出 CPU 执行权，除了可以
让浏览器及时地响应用户的交互，还有其他好处:
分批延时对DOM进行操作，避免一次性操作大量 DOM 节点，可以得到更好的用户体验；
给浏览器一点喘息的机会，它会对代码进行编译优化（JIT）及进行热代码优化，或者对

reflow
进行修正。
核心思想：Fiber 也称协程或者纤程。它和线程并不一样，协程本身是没有并发或者并行能力的（需要
配合线程），它只是一种控制流程的让出机制。让出 CPU 的执行权，让 CPU 能在这段时间执行其他的
操作。渲染的过程可以被中断，可以将控制权交回浏览器，让位给高优先级的任务，浏览器空闲后再恢复
渲染

面试题 40. 简述对componentWillReceiveProps 的理解？

推荐指数：★★ 试题难度：中级 试题类型：原理题

试题回答参考思路：

该方法当props发生变化时执行，初始化render时不执行，在这个回调函数里面，你可以根据属性的变化，通过调用this.setState()来更新你的组件状态，旧的属性还是可以通过this.props来获取,这
里调用更新状态是安全的，并不会触发额外的render调用。
使用好处：在这个生命周期中，可以在子组件的render函数执行前获取新的props，从而更新子组件自己的state。可以将数据请求放在这里进行执行，需要传的参数则从componentWillReceiveProps(nextProps)中获取。而不必将所有的请求都放在父组件中。于是该请求只会在该组件渲染时才会发出，从而减轻请求负担。
componentWillReceiveProps在初始化render的时候不会执行，它会在Component接受到新的状态(Props)时被触发，一般用于父组件状态更新时子组件的重新渲染。

面试题 41. 哪些方法会触发 React 重新渲染？重新渲染 render 会做些什么？

推荐指数：★★★★ 试题难度：初级 试题类型：原理题

试题回答参考思路：

（1）哪些方法会触发 react 重新渲染?
setState（）方法被调用
setState 是 React 中最常用的命令，通常情况下，执行 setState 会触发 render。但是这里有
个点值得关注，执行 setState 的时候不一定会重新渲染。当 setState 传入 null 时，并不会触
发 render。

```
return />;
}
}
}
// pages/page-a.js
export default withFetching(fetching('science-fiction'))(MovieList);
// pages/page-b.js
```

export default withFetching(fetching('action'))(MovieList);
// pages/page-other.js
export default withFetching(fetching('some-other-type'))(MovieList);
复制代码
class App extends React.Component {
父组件重新渲染
只要父组件重新渲染了，即使传入子组件的 props 未发生变化，那么子组件也会重新渲染，进而触发
render
（2）重新渲染 render 会做些什么?
会对新旧 VNode 进行对比，也就是我们所说的Diff算法。
对新旧两棵树进行一个深度优先遍历，这样每一个节点都会一个标记，在到深度遍历的时候，每遍历
到一和个节点，就把该节点和新的节点树进行对比，如果有差异就放到一个对象里面
遍历差异对象，根据差异的类型，根据对应对规则更新VNode
React 的处理 render 的基本思维模式是每次一有变动就会去重新渲染整个应用。在 Virtual DOM
没有出现之前，最简单的方法就是直接调用 innerHTML。Virtual DOM厉害的地方并不是说它比直接
操作 DOM 快，而是说不管数据怎么变，都会尽量以最小的代价去更新 DOM。React 将 render 函数
返回的虚拟 DOM 树与老的进行比较，从而确定 DOM 要不要更新、怎么更新。当 DOM 树很大时，遍历
state = {
a: 1
};
render() {
console.log("render");
return (

{this.state.a}

onClick={() => {
this.setState({ a: 1 }); // 这里并没有改变 a 的值
}}
>
Click me

this.setState(null)}>setState null

);
}
}
复制代码
两棵树进行各种比对还是相当耗性能的，特别是在顶层 setState 一个微小的修改，默认会

去遍历整棵
树。尽管 React 使用高度优化的 Diff 算法，但是这个过程仍然会损耗性能

面试题 42. 简述为什么React并不推荐优先考虑使用Context?

推荐指数：★★★★ 试题难度：中级 试题类型：原理题

试题回答参考思路：

Context目前还处于实验阶段，可能会在后面的发行版本中有很大的变化，事实上这种情况已经发生了，所以为了避免给今后升级带来大的影响和麻烦，不建议在app中使用context。
尽管不建议在app中使用context，但是独有组件而言，由于影响范围小于app，如果可以做到高内聚，不破坏组件树之间的依赖关系，可以考虑使用context
对于组件之间的数据通信或者状态管理，有效使用props或者state解决，然后再考虑使用第三方的成熟库进行解决，以上的方法都不是最佳的方案的时候，在考虑context。
context的更新需要通过setState()触发，但是这并不是很可靠的，Context支持跨组件的访问，但是如果中间的子组件通过一些方法不影响更新，比如shouldComponentUpdate() 返回false那么不能保证Context的更新一定可以使用Context的子组件，因此，Context的可靠性需要关注

面试题 43. React中的setState批量更新的过程是什么?

推荐指数：★★★★ 试题难度：初级 试题类型：原理题

试题回答参考思路：

调用 setState 时，组件的 state 并不会立即改变， setState 只是把要修改的 state 放入一个队列， React 会优化真正的执行时机，并出于性能原因，会将 React 事件处理程序中的多次React 事件处理程序中的多次 setState 的状态修改合并成一次状态修改。 最终更新只产生一次组件及其子组件的重新渲染，这对于大型应用程序中的性能提升至关重要

面试题 44. 简述React中setState的第二个参数作用是什么 ？

推荐指数：★★★★★ 试题难度：初级 试题类型：原理题

试题回答参考思路：

setState 的第二个参数是一个可选的回调函数。这个回调函数将在组件重新渲染后执行。等价于在componentDidUpdate 生命周期内执行。通常建议使用 componentDidUpdate 来代替此方式。在这个回调函数中你可以拿到更新后 state 的值：

```
this.setState({
key1: newState1,
key2: newState2,
...
}, callback) // 第二个参数是 state 更新完成后的回调函数
```

面试题 45. 简述React中的setState和replaceState的区别是什么 ？

推荐指数： ★★★★ 试题难度： 高难 试题类型： 原理题

试题回答参考思路：

（1）setState() setState()用于设置状态对象，其语法如下：
nextState，将要设置的新状态，该状态会和当前的state合并
callback，可选参数，回调函数。该函数会在setState设置成功，且组件重新渲染后调用。
合并nextState和当前state，并重新渲染组件。setState是React事件处理函数中和请求回调函数中
触发UI更新的主要方法。
var ShowTitle = React.createClass({
getDefaultProps:function(){
return{
title : "React"
}
},
render : function(){
return

{this.props.title}

}
});
复制代码
this.setState({
key1: newState1,
key2: newState2,
...
}, callback) // 第二个参数是 state 更新完成后的回调函数
复制代码
setState(object nextState[, function callback])
复制代码
（2）replaceState() replaceState()方法与setState()类似，但是方法只会保留nextState中
状态，原state不在nextState中的状态都会被删除。其语法如下：
nextState，将要设置的新状态，该状态会替换当前的state。
callback，可选参数，回调函数。该函数会在replaceState设置成功，且组件重新渲染后调用。
总结： setState 是修改其中的部分状态，相当于 Object.assign，只是覆盖，不会减少原来的状
态。而replaceState 是完全替换原来的状态，相当于赋值，将原来的 state 替换为另一个对象，如
果新状态属性减少，那么 state 中就没有这个状态

面试题 46. 简述React中的props为什么是只读的？

推荐指数： ★★★ 试题难度： 初级 试题类型： 原理题

试题回答参考思路：

this.props是组件之间沟通的一个接口，原则上来讲，它只能从父组件流向子组件。React具有浓重的函数式编程的思想。
提到函数式编程就要提一个概念：纯函数。它有几个特点：
给定相同的输入，总是返回相同的输出。
过程没有副作用。
不依赖外部状态。
this.props就是汲取了纯函数的思想。props的不可以变性就保证的相同的输入，页面显示的内容是一样的，并且不会产生副作用

面试题 47. 在React中组件的props改变时更新组件的有哪些方法？

推荐指数： ★★★★ 试题难度： 高难 试题类型： 原理题

试题回答参考思路：

在一个组件传入的props更新时重新渲染该组件常用的方法是在componentWillReceiveProps中将新
的props更新到组件的state中（这种state被成为派生状态（Derived State）），从而实现重新渲
染。React 16.3中还引入了一个新的钩子函数getDerivedStateFromProps来专门实现这一需求。
（1）componentWillReceiveProps（已废弃）
在react的componentWillReceiveProps(nextProps)生命周期中，可以在子组件的render函数执
行前，通过this.props获取旧的属性，通过nextProps获取新的props，对比两次props是否相同，从
而更新子组件自己的state。
这样的好处是，可以将数据请求放在这里进行执行，需要传的参数则从componentWillReceiveProps(nextProps)中获取。而不必将所有的请求都放在父组件中。于是该请
求只会在该组件渲染时才会发出，从而减轻请求负担。
（2）getDerivedStateFromProps（16.3引入）
这个生命周期函数是为了替代componentWillReceiveProps存在的，所以在需要使用componentWillReceiveProps时，就可以考虑使用getDerivedStateFromProps来进行替代。
两者的参数是不相同的，而getDerivedStateFromProps是一个静态函数，也就是这个函数不能通过
this访问到class的属性，也并不推荐直接访问属性。而是应该通过参数提供的nextProps以及
prevState来进行判断，根据新传入的props来映射到state。
需要注意的是，如果props传入的内容不需要影响到你的state，那么就需要返回一个null，这个返回值
是必须的，所以尽量将其写到函数的末尾：

12. React中怎么检验props？验证props的目的是什么？
React为我们提供了PropTypes以供验证使用。当我们向Props传入的数据无效（向Props传入的数据类
型和验证的数据类型不符）就会在控制台发出警告信息。它可以避免随着应用越来越复杂从而出现的问
题。并且，它还可以让程序变得更易读。

```
static getDerivedStateFromProps(nextProps, prevState) {
const {type} = nextProps;
// 当传入的type发生变化的时候，更新state
if (type !== prevState.type) {
return {
type,
};
}
// 否则，对于state不进行任何操作
return null;
}
```

面试题 48. 简述React中怎么检验props？验证props的目的是什么？

推荐指数：★★★★ 试题难度：中级 试题类型：原理题

试题回答参考思路：

React为我们提供了PropTypes以供验证使用。当我们向Props传入的数据无效（向Props传入的数据类型和验证的数据类型不符）就会在控制台发出警告信息。它可以避免随着应用越来越复杂从而出现的问题。并且，它还可以让程序变得更易读。

```
import PropTypes from 'prop-types';
class Greeting extends React.Component {
render() {
return (
<h1>Hello, {this.props.name}</h1>
);
}
}
Greeting.propTypes = {
name: PropTypes.string
};
```

当然，如果项目汇中使用了TypeScript，那么就可以不用PropTypes来校验，而使用TypeScript定义接口来校验props。

面试题 49. 简述React 废弃了哪些生命周期？为什么？

推荐指数：★★★ 试题难度：中级 试题类型：原理题

试题回答参考思路：

被废弃的三个函数都是在render之前，因为fber的出现，很可能因为高优先级任务的出现而打断现有任
务导致它们会被执行多次。另外的一个原因则是，React想约束使用者，好的框架能够让人不得已写出容
易维护和扩展的代码，这一点又是从何谈起，可以从新增加以及即将废弃的生命周期分析入手
1) componentWillMount
首先这个函数的功能完全可以使用componentDidMount和 constructor来代替，异步获取的数据的情
况上面已经说明了，而如果抛去异步获取数据，其余的即是初始化而已，这些功能都可以在constructor中执行，除此之外，如果在 willMount 中订阅事件，但在服务端这并不会执行willUnMount事件，也就是说服务端会导致内存泄漏所以componentWillMount完全可以不使用，但使
用者有时候难免因为各 种各样的情况在 componentWilMount中做一些操作，那么React为了约束开发
者，干脆就抛掉了这个API
2) componentWillReceiveProps
在老版本的 React 中，如果组件自身的某个 state 跟其 props 密切相关的话，一直都没有一种很
优雅的处理方式去更新 state，而是需要在 componentWilReceiveProps 中判断前后两个props
是否相同，如果不同再将新的 props更新到相应的 state 上去。这样做一来会破坏 state数据的单
一数据源，导致组件状态变得不可预测，另一方面也会增加组件的重绘次数。类似的业务需求也有很多，
如一个可以横向滑动的列表，当前高亮的 Tab 显然隶属于列表自身的时，根据传入的某个值，直接定位
到某个 Tab。为了解决这些问题，React引入了第一个新的生命周期：
getDerivedStateFromProps。它有以下的优点：
getDSFP是静态方法，在这里不能使用this，也就是一个纯函数，开发者不能写出副作用的代码
开发者只能通过prevState而不是prevProps来做对比，保证了state和props之间的简单关系以
及不需要处理第一次渲染时prevProps为空的情况
基于第一点，将状态变化（setState）和昂贵操作（tabChange）区分开，更加便于render 和
commit 阶段操作或者说优化。
3) componentWillUpdate
与 componentWillReceiveProps 类似，许多开发者也会在 componentWillUpdate 中根据
props 的变化去触发一些回调 。 但不论是 componentWilReceiveProps 还 是
componentWilUpdate，都有可能在一次更新中被调用多次，也就是说写在这里的回调函数也有可能会
被调用多次，这显然是不可取的。与 componentDidMount 类 似，componentDidUpdate 也不存

在这样的问题，一次更新中 componentDidUpdate 只会被调用一次，所以将原先写在 componentWillUpdate 中的回调迁移至 componentDidUpdate 就可以解决这个问题。
另外一种情况则是需要获取DOM元素状态，但是由于在fber中，render可打断，可能在wilMount中获
取到的元素状态很可能与实际需要的不同，这个通常可以使用第二个新增的生命函数的解决
getSnapshotBeforeUpdate(prevProps, prevState)
4) getSnapshotBeforeUpdate(prevProps, prevState)
返回的值作为componentDidUpdate的第三个参数。与willMount不同的是，
getSnapshotBeforeUpdate会在最终确定的render执行之前执行，也就是能保证其获取到的元素状态
与didUpdate中获取到的元素状态相同。官方参考代码：

```
class ScrollingList extends React.Component {
constructor(props) {
super(props);
this.listRef = React.createRef();
}
getSnapshotBeforeUpdate(prevProps, prevState) {
// 我们是否在 list 中添加新的 items ？
// 捕获滚动位置以便我们稍后调整滚动位置。
if (prevProps.list.length < this.props.list.length) {
const list = this.listRef.current;
return list.scrollHeight - list.scrollTop;
}
return null;
}
componentDidUpdate(prevProps, prevState, snapshot) {
// 如果我们 snapshot 有值，说明我们刚刚添加了新的 items，
// 调整滚动位置使得这些新 items 不会将旧的 items 推出视图。
// （这里的 snapshot 是 getSnapshotBeforeUpdate 的返回值）
if (snapshot !== null) {
const list = this.listRef.current;
list.scrollTop = list.scrollHeight - snapshot;
}
}
render() {
return (
 <div ref={this.listRef}>{/* ...contents... */}</div>
```

面试题 50. React 16.X 中 props 改变后在哪个生命周期中处理？

推荐指数：★★★★★ 试题难度：中级 试题类型：原理题

试题回答参考思路：

在getDerivedStateFromProps中进行处理。
这个生命周期函数是为了替代componentWillReceiveProps存在的，所以在需要使

用componentWillReceiveProps时，就可以考虑使用getDerivedStateFromProps来进行替代。
两者的参数是不相同的，而getDerivedStateFromProps是一个静态函数，也就是这个函数不能通过
this访问到class的属性，也并不推荐直接访问属性。而是应该通过参数提供的nextProps以及
prevState来进行判断，根据新传入的props来映射到state。
需要注意的是，如果props传入的内容不需要影响到你的state，那么就需要返回一个null，这个返回值
是必须的，所以尽量将其写到函数的末尾：

```
static getDerivedStateFromProps(nextProps, prevState) {
const {type} = nextProps;
// 当传入的type发生变化的时候，更新state
if (type !== prevState.type) {
return {
type,
};
}
// 否则，对于state不进行任何操作
return null;
}
```

面试题 51. React 性能优化在哪个生命周期？它优化的原理是什么？

推荐指数：★★★★★ 试题难度：高难 试题类型：原理题

试题回答参考思路：

react的父级组件的render函数重新渲染会引起子组件的render方法的重新渲染。但是，有的时候子组件的接受父组件的数据没有变动。子组件render的执行会影响性能，这时就可以使用shouldComponentUpdate来解决这个问题。
使用方法如下

```
shouldComponentUpdate(nexrProps) {
if (this.props.num === nexrProps.num) {
return false
}
return true;
}
```

shouldComponentUpdate提供了两个参数nextProps和nextState，表示下一次props和一次state
的值，当函数返回false时候，render()方法不执行，组件也就不会渲染，返回true时，组件照常重渲
染。此方法就是拿当前props中值和下一次props中的值进行对比，数据相等时，返回false，反之返回
true。
需要注意，在进行新旧对比的时候，是浅对比，也就是说如果比较的数据时引用数据类型，

只要数据的引
用的地址没变，即使内容变了，也会被判定为true。
面对这个问题，可以使用如下方法进行解决：
（1）使用setState改变数据之前，先采用ES6中assgin
进行拷贝，但是assgin只深拷贝的数据的第一层，所以说不是最完美的解决办法：
（2）使用JSON.parse(JSON.stringfy())进行深拷贝，但是遇到数据为undefined和函数时就会
错。

```
shouldComponentUpdate(nexrProps) {
if (this.props.num === nexrProps.num) {
return false
}
return true;
}

const o2 = Object.assign({},this.state.obj)
o2.student.count = '00000';
this.setState({
obj: o2,
})

const o2 = JSON.parse(JSON.stringify(this.state.obj))
o2.student.count = '00000';
this.setState({
obj: o2,
})
```

面试题 52. 简述state 和 props 触发更新的生命周期分别有什么区别？？

推荐指数：★★★★ 试题难度：中级 试题类型：原理题

试题回答参考思路：

state 更新流程 这个过程当中涉及的函数：
1. shouldComponentUpdate: 当组件的 state 或 props 发生改变时，都会首先触发这个生命周
期函数。它会接收两个参数：nextProps, nextState——它们分别代表传入的新 props 和新的
state 值。拿到这两个值之后，我们就可以通过一些对比逻辑来决定是否有 re-render（重渲
染）的必要了。如果该函数的返回值为 false，则生命周期终止，反之继续；
注意：此方法仅作为性能优化的方式而存在。不要企图依靠此方法来“阻止”渲染，因为这可能会产
生 bug。应该考虑使用内置的 PureComponent 组件，而不是手动编写
shouldComponentUpdate()
1. componentWillUpdate：当组件的 state 或 props 发生改变时，会在渲染之前调用
componentWillUpdate。componentWillUpdate 是 React16 废弃的三个生命周期之

一。过
去，我们可能希望能在这个阶段去收集一些必要的信息（比如更新前的 DOM 信息等等），现在我们
完全可以在 React16 的 getSnapshotBeforeUpdate 中去做这些事；
2. componentDidUpdate：componentDidUpdate() 会在UI更新后会被立即调用。它接收
prevProps（上一次的 props 值）作为入参，也就是说在此处我们仍然可以进行 props 值对比
（再次说明 componentWillUpdate 确实鸡肋哈）

props 更新流程:
相对于 state 更新，props 更新后唯一的区别是增加了对 componentWillReceiveProps 的调
用。关于 componentWillReceiveProps，需要知道这些事情：
componentWillReceiveProps：它在Component接受到新的 props 时被触发。
componentWillReceiveProps 会接收一个名为 nextProps 的参数（对应新的 props 值）。
该生命周期是 React16 废弃掉的三个生命周期之一。在它被废弃前，可以用它来比较
this.props 和 nextProps 来重新setState。在 React16 中，用一个类似的新生命周期
getDerivedStateFromProps 来代替它

面试题 53. 简述React中发起网络请求应该在哪个生命周期中进行？为什么？

推荐指数： ★★★★ 试题难度： 中级 试题类型： 原理题

试题回答参考思路：

对于异步请求，最好放在componentDidMount中去操作，对于同步的状态改变，可以放在componentWillMount中，一般用的比较少。
如果认为在componentWillMount里发起请求能提早获得结果，这种想法其实是错误的，通常
componentWillMount比componentDidMount早不了多少微秒，网络上任何一点延迟，这一点差异都
可忽略不计。

react的生命周期： constructor() -> componentWillMount() -> render() -> componentDidMount()
上面这些方法的调用是有次序的，由上而下依次调用。
constructor被调用是在组件准备要挂载的最开始，此时组件尚未挂载到网页上。
componentWillMount方法的调用在constructor之后，在render之前，在这方法里的代码调用
setState方法不会触发重新render，所以它一般不会用来作加载数据之用。
componentDidMount方法中的代码，是在组件已经完全挂载到网页上才会调用被执行，所以可以保
证数据的加载。此外，在这方法中调用setState方法，会触发重新渲染。所以，官方设计这个方法
就是用来加载外部数据用的，或处理其他的副作用代码。与组件上的数据无关的加载，也可

以在
constructor里做，但constructor是做组件state初始化工作，并不是做加载数据这工作的，
constructor里也不能setState，还有加载的时间太长或者出错，页面就无法加载出来。所以有副
作用的代码都会集中在componentDidMount方法里。
总结：
跟服务器端渲染（同构）有关系，如果在componentWillMount里面获取数据，fetch data会执
行两次，一次在服务器端一次在客户端。在componentDidMount中可以解决这个问题，
componentWillMount同样也会render两次。
在componentWillMount中fetch data，数据一定在render后才能到达，如果忘记了设置初始状
态，用户体验不好。
react16.0以后，componentWillMount可能会被执行多次。

面试题 54. 简述非嵌套关系组件的通信方式？

推荐指数：★★★ 试题难度：初级 试题类型：原理题

试题回答参考思路：

即没有任何包含关系的组件，包括兄弟组件以及不在同一个父级中的非兄弟组件。
可以使用自定义事件通信（发布订阅模式）
可以通过redux等进行全局状态管理
如果是兄弟组件通信，可以找到这两个兄弟节点共同的父节点，结合父子间通信方式进行通信

面试题 55. 简述如何解决 props 层级过深的问题？

推荐指数：★★ 试题难度：初级 试题类型：原理题

试题回答参考思路：

使用Context API：提供一种组件之间的状态共享，而不必通过显式组件树逐层传递props；
使用Redux等状态库。

面试题 56. 简述React-Router的实现原理是什么？

推荐指数：★★★★★ 试题难度：中级 试题类型：原理题

试题回答参考思路：

客户端路由实现的思想：
基于 hash 的路由：通过监听
事件，感知 hash 的变化

改变 hash 可以直接通过 location.hash=xxx
基于 H5 history 路由：
改变 url 可以通过 history.pushState 和 resplaceState 等，会将URL压入堆栈，同时能够应用 history.go() 等 API
监听 url 的变化可以通过自定义事件触发实现
react-router 实现的思想：
基于 history 库来实现上述不同的客户端路由实现思想，并且能够保存历史记录等，磨平浏览器
差异，上层无感知
通过维护的列表，在每次 URL 发生变化的回收，通过配置的 路由路径，匹配到对应的 Component，并且 render

面试题 57. 简述React-Router怎么设置重定向?

推荐指数： ★★★ 试题难度： 中级 试题类型： 原理题

试题回答参考思路：

使用组件实现路由的重定向：

当请求 /users/:id 被重定向去 '/users/profile/:id'：
属性 from: string：需要匹配的将要被重定向路径。
属性 to: string：重定向的 URL 字符串
属性 to: object：重定向的 location 对象
属性 push: bool：若为真，重定向操作将会把新地址加入到访问历史记录里面，并且无法回退到
前面的页面。

面试题 58. 简述 react-router 里的 Link 标签和 a 标签的区别？

推荐指数： ★★★ 试题难度： 中级 试题类型： 原理题

试题回答参考思路：

从最终渲染的 DOM 来看，这两者都是链接，都是 标签，区别是：是react-router 里实现路由跳转的链接，一般配合 使用，react-router接管了其默认的链接跳转行为，区别于传统
的页面跳转， 的“跳转”行为只会触发相匹配的 对应的页面内容更新，而不会刷新整个页面。
做了3件事情:
有onclick那就执行onclick
click的时候阻止a标签默认事件
根据跳转href(即是to)，用history (web前端路由两种方式之一，history & hash)跳转，此

```
// location = { pathname: '/react' }

React

// React
```

时只是链接变了，并没有刷新页面而标签就是普通的超链接了，用于从当前页面跳转到href指
向的另一 个页面(非锚点情况)。
a标签默认事件禁掉之后做了什么才实现了跳转?

```
let domArr = document.getElementsByTagName('a')
[...domArr].forEach(item=>{
item.addEventListener('click',function () {
location.href = this.href
})
})
```

面试题 59. 简述React-Router如何获取URL的参数和历史对象？

推荐指数：★★★★ 试题难度： 中级 试题类型： 原理题

试题回答参考思路：

（1）获取URL的参数
get传值
路由配置还是普通的配置，如：'admin'，传参方式如：'admin?id='1111''。通过this.props.location.search获取url获取到一个字符串'?id='1111' 可以用url，qs，querystring，浏览器提供的api URLSearchParams对象或者自己封装的方法去解析出id的值。
动态路由传值
路由需要配置成动态路由：如path='/admin/:id'，传参方式，如'admin/111'。通过this.props.match.params.id 取得url中的动态路由id部分的值，除此之外还可以通过useParams（Hooks）来获取
通过query或state传值
传参方式如：在Link组件的to属性中可以传递对象{pathname:'/admin',query:'111',state:'111'};。通过this.props.location.state或this.props.location.query来获取即可，传递的参数可以是对象、数组等，但是存在缺点就是只
要刷新页面，参数就会丢失。
（2）获取历史对象
如果React >= 16.8 时可以使用 React Router中提供的Hooks

```
let domArr = document.getElementsByTagName('a')
[...domArr].forEach(item=>{
item.addEventListener('click',function () {
```

```
location.href = this.href
})
})
import { useHistory } from "react-router-dom";
let history = useHistory()
2.使用this.props.history获取历史对象
let history = this.props.history;
```

面试题 60. 简述React-Router 4怎样在路由变化时重新渲染同一个组件？

推荐指数：★★★★★ 试题难度：中级 试题类型：原理题

试题回答参考思路：

当路由变化时，即组件的props发生了变化，会调用componentWillReceiveProps等生命周期钩子。那需要做的只是： 当路由改变时，根据路由，也去请求数据

```
class NewsList extends Component {
componentDidMount () {
this.fetchData(this.props.location);
}

fetchData(location) {
const type = location.pathname.replace('/', '') || 'top'
this.props.dispatch(fetchListData(type))
}
componentWillReceiveProps(nextProps) {
if (nextProps.location.pathname != this.props.location.pathname) {
this.fetchData(nextProps.location);
}
}
render () {
...
}
}利用生命周期componentWillReceiveProps，进行重新render的预处理操作
```

面试题 61. 简述React-Router的路由有几种模式？

推荐指数：★★★★★ 试题难度：中级 试题类型：原理题

试题回答参考思路：

React-Router 支持使用 hash（对应 HashRouter）和 browser（对应 BrowserRouter） 两种
路由规则， react-router-dom 提供了 BrowserRouter 和 HashRouter 两个组件来实现应用的
UI 和 URL 同步：

BrowserRouter 创建的 URL 格式：xxx.com/path
HashRouter 创建的 URL 格式：xxx.com/#/path
（1）BrowserRouter
它使用 HTML5 提供的 history API（pushState、replaceState 和 popstate 事件）来保持
UI 和 URL 的同步。由此可以看出，BrowserRouter 是使用 HTML 5 的 history API 来控制路
由跳转的
basename={string}
forceRefresh={bool}
getUserConfirmation={func}
keyLength={number}
/>
其中的属性如下：
basename 所有路由的基准 URL。basename 的正确格式是前面有一个前导斜杠，但不能有尾部斜
杠；

forceRefresh 如果为 true，在导航的过程中整个页面将会刷新。一般情况下，只有在不支持
HTML5 history API 的浏览器中使用此功能；
getUserConfirmation 用于确认导航的函数，默认使用 window.confirm。例如，当从 /a 导
航至 /b 时，会使用默认的 confirm 函数弹出一个提示，用户点击确定后才进行导航，否则不做
任何处理；

```
// 这是默认的确认函数
const getConfirmation = (message, callback) => {
const allowTransition = window.confirm(message);
callback(allowTransition);
}
```

（2）HashRouter
使用 URL 的 hash 部分（即 window.location.hash）来保持 UI 和 URL 的同步。由此可以看
出，HashRouter 是通过 URL 的 hash 属性来控制路由跳转的：
basename={string}
getUserConfirmation={func}
hashType={string}
/>

其参数如下：
basename, getUserConfirmation 和 BrowserRouter 功能一样；

hashType window.location.hash 使用的 hash 类型，有如下几种：
slash – 后面跟一个斜杠，例如 #/ 和 #/sunshine/lollipops；
noslash – 后面没有斜杠，例如 # 和 #sunshine/lollipops；
hashbang – Google 风格的 ajax crawlable，例如 #!/ 和 #!/sunshine/lollipops。

面试题 62. 简述Redux 怎么实现属性传递，介绍下原理？

推荐指数：★★★★ 试题难度：中级 试题类型：原理题

试题回答参考思路：

react-redux 数据传输：view-->action-->reducer-->store-->view。看下点击事件的数据是
如何通过redux传到view上：
view 上的AddClick 事件通过mapDispatchToProps 把数据传到action ---> click:()=>dispatch(ADD)
action 的ADD 传到reducer上
reducer传到store上 const store = createStore(reducer);
store再通过 mapStateToProps 映射穿到view上text:State.text
代码示例：

```
import React from 'react';
import ReactDOM from 'react-dom';
import { createStore } from 'redux';
import { Provider, connect } from 'react-redux';
class App extends React.Component{
render(){
let { text, click, clickR } = this.props;
return(


数据:已有人{text}

加人

减人



)
}
}
const initialState = {
text:5
}
const reducer = function(state,action){
switch(action.type){
case 'ADD':
return {text:state.text+1}
```

```
case 'REMOVE':
return {text:state.text-1}
default:
return initialState;
}
}
let ADD = {
type:'ADD'
}
let Remove = {
type:'REMOVE'
}
const store = createStore(reducer);
let mapStateToProps = function (state){
return{
text:state.text
}
}

let mapDispatchToProps = function(dispatch){
return{
click:()=>dispatch(ADD),
clickR:()=>dispatch(Remove)
}
}
const App1 = connect(mapStateToProps,mapDispatchToProps)(App);
ReactDOM.render(


,document.getElementById('root')
)
```

面试题 63. Redux 中间件是什么？接受几个参数？柯里化函数两端的参数具体是什么？

推荐指数：★★★★★ 试题难度：中级 试题类型：原理题

试题回答参考思路：

Redux 的中间件提供的是位于 action 被发起之后，到达 reducer 之前的扩展点，换而言之，原本view –→> action –> reducer –> store 的数据流加上中间件后变成了 view –> action –>middleware –> reducer –> store ，在这一环节可以做一些"副作用"的操作，如异步请求、打印
日志等。
applyMiddleware源码

```
export default function applyMiddleware(...middlewares) {
return createStore => (...args) => {
// 利用传入的createStore和reducer和创建一个store
```

```
const store = createStore(...args)
let dispatch = () => {
throw new Error()
}
const middlewareAPI = {
getState: store.getState,
dispatch: (...args) => dispatch(...args)
}
// 让每个 middleware 带着 middlewareAPI 这个参数分别执行一遍
const chain = middlewares.map(middleware =>
middleware(middlewareAPI))
// 接着 compose 将 chain 中的所有匿名函数，组装成一个新的函数，即新的
dispatch
dispatch = compose(...chain)(store.dispatch)
return {
...store,
dispatch
}
}
```

}从applyMiddleware中可以看出：
redux中间件接受一个对象作为参数，对象的参数上有两个字段 dispatch 和 getState，分别代
表着 Redux Store 上的两个同名函数。
柯里化函数两端一个是 middewares，一个是store.dispatch

面试题 64. Redux 请求中间件如何处理并发？

推荐指数：★★★★★ 试题难度：高难 试题类型：原理题

试题回答参考思路：

使用redux-Saga redux-saga是一个管理redux应用异步操作的中间件，用于代替
redux-thunk
的。它通过创建 Sagas 将所有异步操作逻辑存放在一个地方进行集中处理，以此将react中
的同步操作
与异步操作区分开来，以便于后期的管理与维护。 redux-saga如何处理并发：
takeEvery
可以让多个 saga 任务并行被 fork 执行。

```
import {
fork,
take
} from "redux-saga/effects"
const takeEvery = (pattern, saga, ...args) => fork(function*() {
while (true) {
const action = yield take(pattern)
yield fork(saga, ...args.concat(action))
}
```

```
})

takeLatest
takeLatest 不允许多个 saga 任务并行地执行。一旦接收到新的发起的 action，它就会取消前面
所有 fork 过的任务（如果这些任务还在执行的话）。 在处理 AJAX 请求的时候，如果只希望获取最
后那个请求的响应， takeLatest 就会非常有用。

import {
cancel,
fork,
take
} from "redux-saga/effects"
const takeLatest = (pattern, saga, ...args) => fork(function*() {
let lastTask
while (true) {
const action = yield take(pattern)
if (lastTask) {
yield cancel(lastTask) // 如果任务已经结束，则 cancel 为空操作
}
lastTask = yield fork(saga, ...args.concat(action))
}
})
```

面试题 65. 简述Redux 状态管理器和变量挂载到 window 中有什么区别？

推荐指数：★★★ 试题难度：中级 试题类型：原理题

试题回答参考思路：

两者都是存储数据以供后期使用。但是Redux状态更改可回溯——Time travel，数据多了的时候可以很清晰的知道改动在哪里发生，完整的提供了一套状态管理模式。
随着 JavaScript 单页应用开发日趋复杂，JavaScript 需要管理比任何时候都要多的 state （状态）。 这些 state 可能包括服务器响应、缓存数据、本地生成尚未持久化到服务器的数据，也包括UI状态，如激活的路由，被选中的标签，是否显示加载动效或者分页器等等。
管理不断变化的 state 非常困难。如果一个 model 的变化会引起另一个 model 变化，那么当view 变化时，就可能引起对应 model 以及另一个model 的变化，依次地，可能会引起另一个 view的变化。直至你搞不清楚到底发生了什么。state 在什么时候，由于什么原因，如何变化已然不受控
制。 当系统变得错综复杂的时候，想重现问题或者添加新功能就会变得举步维艰。 如果这还不够糟糕，考虑一些来自前端开发领域的新需求，如更新调优、服务端渲染、路由跳转前请求数据等等。前端开发者正在经受前所未有的复杂性，难道就这么放弃了吗?当然不是。
这里的复杂性很大程度上来自于：我们总是将两个难以理清的概念混淆在一起：变化和异步。 可以称它们为曼妥思和可乐。如果把二者分开，能做的很好，但混到一起，就变得一团糟。一些库如 React 视图在视图层禁止异步和直接操作 DOM来解决这个问题。美中不足的

是，React 依旧把处理 state 中数据的问题留给了你。Redux就是为了帮你解决这个问题。

面试题 66. 简述mobox 和 redux 有什么区别？

推荐指数：★★★ 试题难度：高难 试题类型：原理题

试题回答参考思路：

（1）共同点
为了解决状态管理混乱，无法有效同步的问题统一维护管理应用状态;
某一状态只有一个可信数据来源（通常命名为store，指状态容器）;
操作更新状态方式统一，并且可控（通常以action方式提供更新状态的途径）;
支持将store与React组件连接，如react-redux，mobx- react;
（2）区别 Redux更多的是遵循Flux模式的一种实现，是一个 JavaScript库，它关注点主要是以下几
方面：
Action：一个JavaScript对象，描述动作相关信息，主要包含type属性和payload属性：
Reducer：定义应用状态如何响应不同动作（action），如何更新状态;
Store：管理action和reducer及其关系的对象，主要提供以下功能
o 维护应用状态并支持访问状态(getState());
o 支持监听action的分发，更新状态(dispatch(action));
o 支持订阅store的变更(subscribe(listener))

异步流：由于Redux所有对store状态的变更，都应该通过action触发，异步任务（通常都是业务或获取数据任务）也不例外，而为了不将业务或数据相关的任务混入React组件中，就需要使用其他框架配合管理异步任务流程，如redux-thunk，redux-saga等;
Mobx是一个透明函数响应式编程的状态管理库，它使得状态管理简单可伸缩：
Action:定义改变状态的动作函数，包括如何变更状态;
Store：集中管理模块状态（State）和动作(action)
Derivation（衍生）：从应用状态中派生而出，且没有任何其他影响的数据

对比总结：
redux将数据保存在单一的store中，mobx将数据保存在分散的多个store中
redux使用plain object保存数据，需要手动处理变化后的操作;mobx适用observable保存数据，数据变化后自动处理响应的操作
redux使用不可变状态，这意味着状态是只读的，不能直接去修改它，而是应该返回一个新的状态，同时使用纯函数;mobx中的状态是可变的，可以直接对其进行修改mobx相对来说比较简单，在其中有很多的抽象，mobx更多的使用面向对象的编程思维;redux会比较
复杂，因为其中的函数式编程思想掌握起来不是那么容易，同时需要借助一系列的中间件来处理异步和副作用
mobx中有更多的抽象和封装，调试会比较困难，同时结果也难以预测;而redux提供能够进行时间回溯的开发工具，同时其纯函数以及更少的抽象，让调试变得更加的容

面试题 67. 简述Redux 和 Vuex 有什么区别，它们的共同思想？

推荐指数：★★★★★ 试题难度：初级 试题类型：原理题

试题回答参考思路：

（1）Redux 和 Vuex区别
Vuex改进了Redux中的Action和Reducer函数，以mutations变化函数取代Reducer，无需switch，只需在对应的mutation函数里改变state值即可
Vuex由于Vue自动重新渲染的特性，无需订阅重新渲染函数，只要生成新的State即可Vuex数据流的顺序是：View调用store.commit提交对应的请求到Store中对应的mutation函数->store改变（vue检测到数据变化自动渲染）
通俗点理解就是，vuex 弱化 dispatch，通过commit进行 store状态的一次更变；取消了action概念，不必传入特定的 action形式进行指定变更；弱化reducer，基于commit参数直接对数据进行转变，使得框架更加简易

（2）共同思想
单一的数据源
变化可以预测
本质上：redux与vuex都是对mvvm思想的服务，将数据从视图中抽离的一种方案

面试题 68. 简述Redux 中间件是怎么拿到store 和 action? 然后怎么处理？

推荐指数：★★★★★ 试题难度：中级 试题类型：原理题

试题回答参考思路：

redux中间件本质就是一个函数柯里化。redux applyMiddleware Api 源码中每个middleware 接受2个参数， Store 的getState 函数和dispatch 函数，分别获得store和action，最终返回一个函数。该函数会被传入 next 的下一个 middleware 的 dispatch 方法，并返回一个接收 action
的新函数，这个函数可以直接调用 next（action），或者在其他需要的时刻调用，甚至根本不去调用它。调用链中最后一个 middleware 会接受真实的 store的 dispatch 方法作为 next 参数，并借此结束调用链。所以，middleware 的函数签名是（{ getState，dispatch })=> next =>action

面试题 69. 简述为什么 useState 要使用数组而不是对象？

推荐指数：★★★★ 试题难度：高难 试题类型：原理题

试题回答参考思路：

useState 的用法：
可以看到 useState 返回的是一个数组，那么为什么是返回数组而不是返回对象呢?
这里用到了解构赋值，所以先来看一下ES6 的解构赋值：
数组的解构赋值

```
const foo = [1, 2, 3];
const [one, two, three] = foo;
console.log(one); // 1
console.log(two); // 2
console.log(three); // 3
```

对象的解构赋值
const user = {
id: 888,
name: "xiaoxin"
};
const { id, name } = user;
console.log(id); // 888
console.log(name); // "xiaoxin"
看完这两个例子，答案应该就出来了：
如果 useState 返回的是数组，那么使用者可以对数组中的元素命名，代码看起来也比较干净
如果 useState 返回的是对象，在解构对象的时候必须要和 useState 内部实现返回的对象同
名，想要使用多次的话，必须得设置别名才能使用返回值
下面来看看如果 useState 返回对象的情况：
const [count, setCount] = useState(0)
const foo = [1, 2, 3];
const [one, two, three] = foo;
console.log(one); // 1
console.log(two); // 2
console.log(three); // 3
看完这两个例子，答案应该就出来了：
如果 useState 返回的是数组，那么使用者可以对数组中的元素命名，代码看起来也比较干净
如果 useState 返回的是对象，在解构对象的时候必须要和 useState 内部实现返回的对象同
名，想要使用多次的话，必须得设置别名才能使用返回值
下面来看看如果 useState 返回对象的情况

// 第一次使用
const { state, setState } = useState(false);
// 第二次使用
const { state: counter, setState: setCounter } = useState(0)

这里可以看到，返回对象的使用方式还是挺麻烦的，更何况实际项目中会使用的更频繁。 总结：*useState 返回的是 array 而不是 object 的原因就是为了* 降低使用的复杂度，返回数组的
话可以直接根据顺序解构，而返回对象的话要想使用多次就需要定义别名了。

面试题 70. 简述React Hooks 解决了哪些问题？

推荐指数：★★★★ 试题难度：中级 试题类型：原理题

试题回答参考思路：

React Hooks 主要解决了以下问题：
（1）在组件之间复用状态逻辑很难

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）解决此类问题可
以使用 render props 和 高阶组件。但是这类方案需要重新组织组件结构，这可能会很麻烦，并且会
使代码难以理解。由 providers，consumers，高阶组件，render props 等其他抽象层组成的组件
会形成“嵌套地狱”。尽管可以在 DevTools 过滤掉它们，但这说明了一个更深层次的问题：React 需
要为共享状态逻辑提供更好的原生途径。
可以使用 Hook 从组件中提取状态逻辑，使得这些逻辑可以单独测试并复用。Hook 使我们在无需修改
组件结构的情况下复用状态逻辑。 这使得在组件间或社区内共享 Hook 变得更便捷。
（2）复杂组件变得难以理解
在组件中，每个生命周期常常包含一些不相关的逻辑。例如，组件常常在 componentDidMount 和
componentDidUpdate 中获取数据。但是，同一个 componentDidMount 中可能也包含很多其它的
逻辑，如设置事件监听，而之后需在 componentWillUnmount 中清除。相互关联且需要对照修改的代
码被进行了拆分，而完全不相关的代码却在同一个方法中组合在一起。如此很容易产生 bug，并且导致
逻辑不一致。
在多数情况下，不可能将组件拆分为更小的粒度，因为状态逻辑无处不在。这也给测试带来了一定挑战。
同时，这也是很多人将 React 与状态管理库结合使用的原因之一。但是，这往往会引入了很多抽象概
念，需要你在不同的文件之间来回切换，使得复用变得更加困难。
为了解决这个问题，Hook 将组件中相互关联的部分拆分成更小的函数（比如设置订阅或请求数据），而
并非强制按照生命周期划分。你还可以使用 reducer 来管理组件的内部状态，使其更加可预测。
（3）难以理解的 class
除了代码复用和代码管理会遇到困难外，class 是学习 React 的一大屏障。我们必须去理解
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。没有稳
定的语法提案，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。
为了解决这些问题，Hook 使你在非 class 的情况下可以使用更多的 React 特性。 从概念上讲，
React 组件一直更像是函数。而 Hook 则拥抱了函数，同时也没有牺牲 React 的精神原则。Hook
提供了问题的解决方案，无需学习复杂的函数式或响应式编程技术

面试题 71. 简述 React Hook 的使用限制有哪些？

推荐指数：★★★★ 试题难度：高难 试题类型：原理题

试题回答参考思路：

React Hooks 的限制主要有两条：
不要在循环、条件或嵌套函数中调用 Hook；
在 React 的函数组件中调用 Hook。
那为什么会有这样的限制呢？Hooks 的设计初衷是为了改进 React 组件的开发模式。在旧有的开发模式下遇到了三个问题。
组件之间难以复用状态逻辑。过去常见的解决方案是高阶组件、render props 及状态管理框架。
复杂的组件变得难以理解。生命周期函数与业务逻辑耦合太深，导致关联部分难以拆分。
人和机器都很容易混淆类。常见的有 this 的问题，但在 React 团队中还有类难以优化的问题，希望在编译优化层面做出一些改进。
这三个问题在一定程度上阻碍了 React 的后续发展，所以为了解决这三个问题，Hooks 基于函数组件开始设计。然而第三个问题决定了 Hooks 只支持函数组件。
那为什么不要在循环、条件或嵌套函数中调用 Hook 呢？因为 Hooks 的设计是基于数组实现。在调用时按顺序加入数组中，如果使用循环、条件或嵌套函数很有可能导致数组取值错位，执行错误的 Hook。
当然，实质上 React 的源码里不是数组，是链表。
这些限制会在编码上造成一定程度的心智负担，新手可能会写错，为了避免这样的情况，可以引入ESLint 的 Hooks 检查插件进行预防。

面试题 72. 简述useEffect 与 useLayoutEffect 的区别？

推荐指数：★★★ 试题难度：高难 试题类型：原理题

试题回答参考思路：

（1）共同点
运用效果： useEffect 与 useLayoutEffect 两者都是用于处理副作用，这些副作用包括改变DOM、设置订阅、操作定时器等。在函数组件内部操作副作用是不被允许的，所以需要使用这两个函数去处理。
使用方式： useEffect 与 useLayoutEffect 两者底层的函数签名是完全一致的，都是调用的mountEffectImpl方法，在使用上也没什么差异，基本可以直接替换。
（2）不同点
使用场景： useEffect 在 React 的渲染过程中是被异步调用的，用于绝大多数场景；而 useLayoutEffect 会在所有的 DOM 变更之后同步调用，主要用于处理 DOM 操作、调整样式、避免页面闪烁等问题。也正因为是同步处理，所以需要避免在 useLayoutEffect 做计算量较大的
耗时任务从而造成阻塞。
使用效果： useEffect是按照顺序执行代码的，改变屏幕像素之后执行（先渲染，后改变DOM），当改变屏幕内容时可能会产生闪烁；useLayoutEffect是改变屏幕像素之前就执行了（会推迟页面显示的事件，先改变DOM后渲染），不会产生闪烁。useLayoutEffect总是比useEffect先执行。
在未来的趋势上，两个 API 是会长期共存的，暂时没有删减合并的计划，需要开发者根据场

景去自行选择。React 团队的建议非常实用，如果实在分不清，先用 useEffect，一般问题不大；如果页面有异常，再直接替换为 useLayoutEffect 即可。

面试题 73. 简述React diff 算法的原理是什么？

推荐指数：★★★★★ 试题难度：高难 试题类型：原理题

试题回答参考思路：

实际上，diff 算法探讨的就是虚拟 DOM 树发生变化后，生成 DOM 树更新补丁的方式。它通过对比新
旧两株虚拟 DOM 树的变更差异，将更新补丁作用于真实 DOM，以最小成本完成视图更新。
具体的流程如下：
真实的 DOM 首先会映射为虚拟 DOM；
当虚拟 DOM 发生变化后，就会根据差距计算生成 patch，这个 patch 是一个结构化的数据，内
容包含了增加、更新、移除等；
根据 patch 去更新真实的 DOM，反馈到用户的界面上。
一个简单的例子：
这里，首先假定 ExampleComponent 可见，然后再改变它的状态，让它不可见 。映射为真实的 DOM
操作是这样的，React 会创建一个 div 节点。
当把 visbile 的值变为 false 时，就会替换 class 属性为 hidden，并重写内部的 innerText
为 hidden。这样一个生成补丁、更新差异的过程统称为 diff 算法。
diff算法可以总结为三个策略，分别从树、组件及元素三个层面进行复杂度的优化：

```
import React from 'react'
export default class ExampleComponent extends React.Component {
render() {
if(this.props.isVisible) {
return

visbile
;
}
return
hidden
;
}
}
```

策略一：忽略节点跨层级操作场景，提升比对效率。（基于树进行对比）
这一策略需要进行树比对，即对树进行分层比较。树比对的处理手法是非常“暴力”的，即两棵树只对同一
层次的节点进行比较，如果发现节点已经不存在了，则该节点及其子节点会被完全删除掉，不会用于进一
步的比较，这就提升了比对效率。

策略二：如果组件的 class 一致，则默认为相似的树结构，否则默认为不同的树结构。（基于组件进行
对比）
在组件比对的过程中：
如果组件是同一类型则进行树比对；
如果不是则直接放入补丁中。
只要父组件类型不同，就会被重新渲染。这也就是为什么 shouldComponentUpdate、PureComponent 及 React.memo 可以提高性能的原因。
策略三：同一层级的子节点，可以通过标记 key 的方式进行列表对比。（基于节点进行对比）
元素比对主要发生在同层级中，通过标记节点操作生成补丁。节点操作包含了插入、移动、删除等。其中
节点重新排序同时涉及插入、移动、删除三个操作，所以效率消耗最大，此时策略三起到了至关重要的作
用。通过标记 key 的方式，React 可以直接移动 DOM 节点，降低内耗。

面试题 74. 简述 React key 是干嘛用的 为什么要加？key 主要是解决哪一类问题的?

推荐指数：★★★ 试题难度：初级 试题类型：原理题

试题回答参考思路：

Keys 是 React 用于追踪哪些列表中元素被修改、被添加或者被移除的辅助标识。在开发过程中，我们需要保证某个元素的 key 在其同级元素中具有唯一性。
在 React Diff 算法中 React 会借助元素的 Key 值来判断该元素是新近创建的还是被移动而来的元素，从而减少不必要的元素重渲染此外，React 还需要借助 Key 值来判断元素与本地状态的关联关系。
注意事项：
key值一定要和具体的元素一一对应；
尽量不要用数组的index去作为key；
不要在render的时候用随机数或者其他操作给元素加上不稳定的key，这样造成的性能开销比不加key的情况下更糟糕。

面试题 75. 简述虚拟 DOM 的引入与直接操作原生 DOM 相比，哪一个效率更高，为什么？

推荐指数：★★★★ 试题难度：中级 试题类型：原理题

试题回答参考思路：

虚拟DOM相对原生的DOM不一定是效率更高，如果只修改一个按钮的文案，那么虚拟DOM 的操作无论如何都不可能比真实的 DOM 操作更快。在首次渲染大量DOM时，由于多了一层虚拟DOM的计算，虚拟DOM也会比innerHTML插入慢。它能保证性能下限，在真实DOM操作的时候进行针对性的优化时，还是更快的。所以要根据具体的场景进行探讨。
在整个 DOM 操作的演化过程中，其实主要矛盾并不在于性能，而在于开发者写得爽不爽，在于研发体验/研发效率。虚拟 DOM 不是别的，正是前端开发们为了追求更好的研发体验和研发效率而创造出来的高阶产物。虚拟 DOM 并不一定会带来更好的性能，React 官方也从来没有把虚拟 DOM 作为性能层面的卖点对外输出过。虚拟 DOM 的优越之处在于，它能

够在提供更爽、更高效的研发模式（也就是函数式的 UI 编程方式）的同时，仍然保持一个还不错的性能

面试题 76. 简述React 与 Vue 的 diff 算法有何不同？

推荐指数：★★★★★ 试题难度：中级 试题类型：原理题

试题回答参考思路：

diff 算法是指生成更新补丁的方式，主要应用于虚拟 DOM 树变化后，更新真实 DOM。所以 diff 算法一定存在这样一个过程：触发更新 → 生成补丁 → 应用补丁。
React 的 diff 算法，触发更新的时机主要在 state 变化与 hooks 调用之后。此时触发虚拟 DOM树变更遍历，采用了深度优先遍历算法。但传统的遍历方式，效率较低。为了优化效率，使用了分治的方式。将单一节点比对转化为了 3 种类型节点的比对，分别是树、组件及元素，以此提升效率。
树比对：由于网页视图中较少有跨层级节点移动，两株虚拟 DOM 树只对同一层次的节点进行比较。
组件比对：如果组件是同一类型，则进行树比对，如果不是，则直接放入到补丁中。
元素比对：主要发生在同层级中，通过标记节点操作生成补丁，节点操作对应真实的 DOM 剪裁操作。
以上是经典的 React diff 算法内容。自 React 16 起，引入了 Fiber 架构。为了使整个更新过程可随时暂停恢复，节点与树分别采用了 FiberNode 与 FiberTree 进行重构。fiberNode 使用了双链表的结构，可以直接找到兄弟节点与子节点。整个更新过程由 current 与 workInProgress 两
株树双缓冲完成。workInProgress 更新完成后，再通过修改 current 相关指针指向新节点。
Vue 的整体 diff 策略与 React 对齐，虽然缺乏时间切片能力，但这并不意味着 Vue 的性能更差，因为在 Vue 3 初期引入过，后期因为收益不高移除掉了。除了高帧率动画，在 Vue 中其他的场景几乎都可以使用防抖和节流去提高响应性能

面试题 77. React组件命名推荐的方式是哪个？

推荐指数：★★ 试题难度：初级 试题类型：原理题

试题回答参考思路：

通过引用而不是使用来命名组件displayName。
使用displayName命名组件
export default React.createClass({ displayName: 'TodoApp', // ...})
React推荐的方法：
export default class TodoApp extends React.Component { // ...}

面试题 78. 简述 react 最新版本解决了什么问题，增加了哪些东西？

推荐指数：★★★ 试题难度：中级 试题类型：原理题

试题回答参考思路：

React 16.x的三大新特性 Time Slicing、Suspense、 hooksTime Slicing（解决CPU速度问题）使得在执行任务的期间可以随时暂停，跑去干别的事情，这个
特性使得react能在性能极其差的机器跑时，仍然保持有良好的性能
Suspense （解决网络IO问题） 和lazy配合，实现异步加载组件。 能暂停当前组件的渲染， 当完成某件事以后再继续渲染，解决从react出生到现在都存在的「异步副作用」的问题，而且解决得非的优雅，使用的是 T异步但是同步的写法，这是最好的解决异步问题的方式
提供了一个内置函数componentDidCatch，当有错误发生时，可以友好地展示 fallback 组件;
可以捕捉到它的子元素（包括嵌套子元素）抛出的异常; 可以复用错误组件。

（1）React16.8 加入hooks，让React函数式组件更加灵活，hooks之前，React存在很多问题：
在组件间复用状态逻辑很难
复杂组件变得难以理解，高阶组件和函数组件的嵌套过深。
class组件的this指向问题
难以记忆的生命周期
hooks很好的解决了上述问题，hooks提供了很多方法
useState 返回有状态值，以及更新这个状态值的函数
useEffect 接受包含命令式，可能有副作用代码的函数。
useContext 接受上下文对象（从 React.createContext返回的值）并返回当前上下文值，
useReducer useState 的替代方案。接受类型为 （state，action）=> newState的 reducer，并返回与dispatch方法配对的当前状态。
useCalLback 返回一个回忆的memoized版本，该版本仅在其中一个输入发生更改时才会更改。纯
函数的输入输出确定性 o useMemo 纯的一个记忆函数 o useRef 返回一个可变的ref对象，其
Current 属性被初始化为传递的参数，返回的 ref 对象在组件的整个生命周期内保持不变。
useImperativeMethods 自定义使用ref时公开给父组件的实例值
useMutationEffect 更新兄弟组件之前，它在React执行其DOM改变的同一阶段同步触发
useLayoutEffect DOM改变后同步触发。使用它来从DOM读取布局并同步重新渲染

（2）React16.9
重命名 Unsafe 的生命周期方法。新的 UNSAFE_前缀将有助于在代码 review 和 debug 期间，使这些有问题的字样更突出
废弃 javascrip:形式的 URL。以javascript:开头的URL 非常容易遭受攻击，造成安全漏洞。
废弃"Factory"组件。 工厂组件会导致 React 变大且变慢。
act（）也支持异步函数，并且你可以在调用它时使用 await。
使用 进行性能评估。在较大的应用中追踪性能回归可能会很方便

（3）React16.13.0
支持在渲染期间调用setState，但仅适用于同一组件可检测冲突的样式规则并记录警告
废弃 unstable_createPortal，使用CreatePortal将组件堆栈添加到其开发警告中，使开发人员能够隔离bug并调试其程序，这可以清楚地说明问题所在，并更快地定位和修复错误

面试题 79. 简述在React中页面重新加载时怎样保留数据？

推荐指数：★★★★★ 试题难度：中级 试题类型：原理题

试题回答参考思路：

这个问题就设计到了数据持久化， 主要的实现方式有以下几种：
Redux： 将页面的数据存储在redux中，在重新加载页面时，获取Redux中的数据；
data.js： 使用webpack构建的项目，可以建一个文件，data.js，将数据保存data.js中，跳转页面后获取；
sessionStorge： 在进入选择地址页面之前，componentWillUnMount的时候，将数据存储到
sessionStorage中，每次进入页面判断sessionStorage中有没有存储的那个值，有，则读取渲染数据；没有，则说明数据是初始化的状态。返回或进入除了选择地址以外的页面，清掉存储的
sessionStorage，保证下次进入是初始化的数据
history API： History API 的 pushState 函数可以给历史记录关联一个任意的可序列化 state，所以可以在路由 push 的时候将当前页面的一些信息存到 state 中，下次返回到这个
页面的时候就能从 state 里面取出离开前的数据重新渲染。react-router 直接可以支持。这个方法适合一些需要临时存储的场景

面试题 80. 简述在React中怎么使用async/await？

推荐指数：★★★★ 试题难度：初级 试题类型：原理题

试题回答参考思路：

async/await是ES7标准中的新特性。如果是使用React官方的脚手架创建的项目，就可以直接使用。如果是在自己搭建的webpack配置的项目中使用，可能会遇到regeneratorRuntime is notdefined 的异常错误。那么我们就需要引入babel，并在babel中配置使用async/await。可以利用
babel的 transform-async-to-module-method 插件来转换其成为浏览器支持的语法，虽然没有性能的提升，但对于代码编写体验要更好

面试题 81. 简述React.Children.map和js的map有什么区别？

推荐指数：★★★★ 试题难度：初级 试题类型：原理题

试题回答参考思路：

JavaScript 中的 map 不会对为 null 或者 undefined 的数据进行处理，而React.Children.map中的map可以处理React.Children为null或者undefined的情况

面试题 82. 简述为什么 React 要用 JSX？

推荐指数：★★★ 试题难度：高难 试题类型：原理题

试题回答参考思路：

JSX 是一个 JavaScript 的语法扩展，或者说是一个类似于 XML 的 ECMAScript 语法扩展。它本身没有太多的语法定义，也不期望引入更多的标准。
其实 React 本身并不强制使用 JSX。在没有 JSX 的时候，React 实现一个组件依赖于使用
React.createElement 函数。代码如下：

```
class Hello extends React.Component {
render() {
return React.createElement(
'div',
null,
`Hello ${this.props.toWhat}`
);
}
}
ReactDOM.render(
React.createElement(Hello, {toWhat: 'World'}, null),
document.getElementById('root')
);
```

而 JSX 更像是一种语法糖，通过类似 XML 的描述方式，描写函数对象。在采用 JSX 之后，这段代码会这样写：

```
class Hello extends React.Component {
render() {
return

Hello {this.props.toWhat}
;
}
}
ReactDOM.render(
,
document.getElementById('root')
);
```

通过对比，可以清晰地发现，代码变得更为简洁，而且代码结构层次更为清晰。
因为 React 需要将组件转化为虚拟 DOM 树，所以在编写代码时，实际上是在手写一棵结构树。而XML在树结构的描述上天生具有可读性强的优势。
但这样可读性强的代码仅仅是给写程序的同学看的，实际上在运行的时候，会使用 Babel 插件将 JSX语法的代码还原为 React.createElement 的代码。
总结： JSX 是一个 JavaScript 的语法扩展，结构类似 XML。JSX 主要用于声明 React 元素，但React 中并不强制使用 JSX。即使使用了 JSX，也会在构建过程中，通过 Babel 插件编译为React.createElement。所以 JSX 更像是 React.createElement 的一种语法糖。
React 团队并不想引入 JavaScript 本身以外的开发体系。而是希望通过合理的关注点分离保持组件开发的纯粹性。

面试题 83. 简述HOC相比 mixins 有什么优点?

推荐指数: ★★★ 试题难度: 中级 试题类型: 原理题

试题回答参考思路:

HOC 和 Vue 中的 mixins 作用是一致的，并且在早期 React 也是使用 mixins 的方式。但是在使用 class 的方式创建组件以后，mixins 的方式就不能使用了，并且其实 mixins 也是存在一些问题的，比如:
隐含了一些依赖，比如我在组件中写了某个 state 并且在 mixin 中使用了，就这存在了一个依赖关系。万一下次别人要移除它，就得去 mixin 中查找依赖多个 mixin 中可能存在相同命名的函数，同时代码组件中也不能出现相同命名的函数，否则就是重写了，其实我一直觉得命名真的是一件麻烦事。。
雪球效应，虽然我一个组件还是使用着同一个 mixin，但是一个 mixin 会被多个组件使用，可能会存在需求使得 mixin 修改原本的函数或者新增更多的函数，这样可能就会产生一个维护成本HOC 解决了这些问题，并且它们达成的效果也是一致的，同时也更加的政治正确（毕竟更加函数式
了）

面试题 84. 简述React 中的高阶组件运用了什么设计模式 ?

推荐指数: ★★★★ 试题难度: 高难 试题类型: 原理题

试题回答参考思路:

使用了装饰模式，高阶组件的运用

```
function withWindowWidth(BaseComponent) {
class DerivedClass extends React.Component {
state = {
windowWidth: window.innerWidth,
}
onResize = () => {
this.setState({
windowWidth: window.innerWidth,
})
}
componentDidMount() {
window.addEventListener('resize', this.onResize)
}
componentWillUnmount() {
window.removeEventListener('resize', this.onResize);
}
render() {
return
}
}
return DerivedClass;
```

```
}
const MyComponent = (props) => {
return

Window width is: {props.windowWidth}

};
export default withWindowWidth(MyComponent);
```

装饰模式的特点是不需要改变 被装饰对象 本身，而只是在外面套一个外壳接口。
JavaScript 目前已
经有了原生装饰器的提案，其用法如下

```
@testable
class MyTestableClass {
}
```

面试题 85. 如果想要在组件第一次加载后获取该组件的dom元素，应当在以下哪个生命周期中进行？

A：componentDidUpdate()
B：componentDidMount()
C：componentWillUnmount()
D：shouldComponentUpdate()

推荐指数：★★★★ 试题难度：中级 试题类型：选择题

试题回答参考思路：

B

面试题 86. 简述使用哪个Hook可以为函数组件添加state？

A：useEffect
B：useReducer
C：useContext
D：useState

推荐指数：★★★ 试题难度：初级 试题类型：选择题

试题回答参考思路：

D
useState返回一个state，以及更新state的函数，在初始渲染期间，返回的状态（state）与传入的第一个参数（initialState）值相同。setState函数用于更新state，它接收一个新的state值并将组件的一次重新渲染加入队列

面试题 87. 简述在React组件中的JSX里，如果要添加类选择器，可以给DOM节点和组件添加()属性？

A：classStyle
B：class

C：styleName
D：className

推荐指数：★★★★ 试题难度：中级 试题类型：选择题

试题回答参考思路：

D
className属性用于指定CSS的class，此特性适用于所有常规 DOM 节点和 SVG 元素，如

，及其它标签

面试题 88. 简述JSX代码最终会被转换成使用（）方法生成的react元素？

A：React.jsx
B：React.cloneElement
C：React.createElement
D：React.createJsx

推荐指数：★★ 试题难度：初级 试题类型：选择题

试题回答参考思路：

C

面试题 89. 简述以下哪项对react-router-dom中组件分类是错误的？

A：路由器组件: BrowserRouter和HashRouter
B：导航组件: Link和NavLink
C：路由匹配组件: Route和Switch
D：导航组件: Route和Switch

推荐指数：★★ 试题难度：中级 试题类型：选择题

试题回答参考思路：

D
在react-router-dom中通常使用的组件有三种路由器组件：如BrowserRouter和HashRouter
路由匹配组件: Route和Switch 组件
导航组件: Link和NavLink 组件

面试题 90. 简述在函数组件中使用哪个Hook可以包裹副作用（改变 DOM、添加订阅、设置定时器、记录日志等）？

A：useReducer
B：useState
C：useEffect
D：useCallback

推荐指数： ★★★★★ 试题难度： 中级 试题类型： 选择题

试题回答参考思路：

C
使用 useEffect 完成副作用操作。赋值给 useEffect 的函数会在组件渲染到屏幕之后执行。你可以把useEffect 看作从 React 的纯函数式世界通往命令式世界的逃生通道。

面试题 91. 简述以下关于对Router组件设置history属性值解释正确的是？

A：browserHistory，背后调用的是浏览器的History API。
B：hashHistory，路由将通过URL的hash部分（#）切换
C：createMemoryHistory主要用于服务器渲染，它创建一个内存中的history对象，不与浏览器URL互动
D：其他选项都正确

推荐指数： ★★★★★ 试题难度： 初级 试题类型： 选择题

试题回答参考思路：

D

面试题 92. 简述以下不属于React支持的键盘事件的是？

A：onKeyUp
B：onKeyDown
C：onKey
D：onKeyPress

推荐指数： ★★★ 试题难度： 初级 试题类型： 选择题

试题回答参考思路：

C
React支持的键盘事件有onKeyDown，onKeyPress，onKeyUp

面试题 93. 简述在componentWillUnmount()中以下哪个操作是错误的？

A：清除timer
B：取消网络请求
C：调用setState
D：清除在componentDidMount()中创建的订阅

推荐指数： ★★★★ 试题难度： 中级 试题类型： 选择题

试题回答参考思路：

C
componentWillUnmount()中不应调用setState()，因为该组件将永远不会重新渲染。组件

实例卸载后，将永远不会再挂载它。

面试题 94. 简述以下不属于React支持的动画事件的是 ？

A：onAnimationIteration
B：onAnimationStart
C：onAnimationOver
D：onAnimationEnd

推荐指数：★★★★ 试题难度：初级 试题类型：选择题

试题回答参考思路：

C
React支持的动画事件有onAnimationStart，onAnimationEnd，onAnimationIteration

面试题 95. 以下关于Fragments组件说法错误的是?

A：在jsx中使用Fragments组件会在html中生成dom节点
B：Fragments组件可以简写成“<>”
C：React中一个常见模式是为一个组件返回多个元素。Fragments可以让你聚合一个子元素列表
D：Fragments组件中可以设置key属性，但是简写后将不支持

推荐指数：★★★ 试题难度：中级 试题类型：选择题

试题回答参考思路：

A

面试题 96. Class内部有一handleClick方法如下代码块，在点击事件中触发handleClick的正确方法是 ？

```
handleClick(){
console.log('this.state:',this.state);
}
```

A：onClick={ () => this.handleClick()}
B：onClick={ this.handleClick()}
C：onClick={ this.handleClick}
D：onclick={ handleClick}

推荐指数：★★★★ 试题难度：中级 试题类型：选择题

试题回答参考思路：

A
这并不是 React 特有的行为，在JavaScript中 Class的方法默认不会绑定this

面试题 97. 简述以下选项中，哪个不是Hook的优点 ？

A：Hook能覆盖class的所有使用场景
B：Hook使你在无需修改组件结构的情况下复用状态逻辑
C：Hook使你在非class的情况下可以使用更多的React特性
D：Hook将组件中相互关联的部分拆分成更小的函数（比如设置订阅或请求数据）

推荐指数：★★★ 试题难度：初级 试题类型：选择题

试题回答参考思路：

A

面试题 98. 简述以下不是Redux的使用原则的是？

A：单一数据源
B：双向数据流
C：State是只读的
D：使用纯函数来执行修改

推荐指数：★★★ 试题难度：中级 试题类型：选择题

试题回答参考思路：

B
Redux可以用这三个基本原则来描述：1.单一数据源 2.State是只读的 3.使用纯函数来执行修改

面试题 99. 简述下面是对refs使用不合适的是？

A：获取子组件的实例，直接通过子组件的setState方法修改子组件的state
B：管理焦点，文本选择或媒体播放
C：集成第三方 DOM 库
D：触发强制动画

推荐指数：★★★ 试题难度：中级 试题类型：选择题

试题回答参考思路：

A
避免使用refs来做任何可以通过声明式实现来完成的事情，下面是几个适合使用refs的情况：
1.管理焦点，文本选择或媒体播放。
2.触发强制动画。
3.集成第三方 DOM 库

面试题 100. 简述关于Hook中里的useEffect的执行时机，下面说法正确的是？

A：与class组件的componentDidMount相同
B：与class组件的componentWillUpdate相同
C：与class组件的componentDidUpdate相同
D：其他选项都不正确

推荐指数： ★★ 试题难度： 中级 试题类型： 选择题

试题回答参考思路：

D

如果熟悉 React class 的生命周期函数，你可以把 useEffect Hook 看做 componentDidMount，componentDidUpdate 和 componentWillUnmount 这三个函数的组合