社招前端一面必会react面试题集锦

vue 或者react 优化整体优化

1. 虚拟dom

为什么虚拟 dom 会提高性能?(必考)

虚拟 dom 相当于在 js 和真实 dom 中间加了一个缓存，利用 dom diff 算法避免了没有必要的 dom 操作，从而提高性能。

用 JavaScript 对象结构表示 DOM 树的结构；然后用这个树构建一个真正的 DOM 树，插到文档当中当状态变更的时候，重新构造一棵新的对象树。然后用新的树和旧的树进行比较，记录两棵树差异把 2 所记录的差异应用到步骤 1 所构建的真正的 DOM 树上，视图就更新了。

React如何获取组件对应的DOM元素?

可以用ref来获取某个子节点的实例，然后通过当前class组件实例的一些特定属性来直接获取子节点实例。ref有三种实现方法:

- **字符串格式**：字符串格式，这是React16版本之前用得最多的，例如：`<p ref="info">span</p>`
- **函数格式**：ref对应一个方法，该方法有一个参数，也就是对应的节点实例，例如：`<p ref={ele => this.info = ele}></p>`
- **createRef方法**：React 16提供的一个API，使用React.createRef()来实现

React 废弃了哪些生命周期？为什么?

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：

javascript 复制代码

```
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>
    );
  }
}
```

如何使用4.0版本的 React Router?

React Router 4.0版本中对 hashHistory做了迁移，执行包安装命令 npm install react-router-dom后，按照如下代码进行使用即可。

```javascript 复制代码
import { HashRouter, Route, Redirect, Switch } from " react-router-dom";
class App extends Component {
  render() {
    return (
      <div>
        <Switch>
          <Route path="/list" componen t={List}></Route>
          <Route path="/detail/: id" component={Detail}>
            {" "}
          </Route>
          <Redirect from="/ " to="/list">
            {" "}
          </Redirect>
        </Switch>
      </div>
    );
  }
}
const routes = (
  <HashRouter>
    <App> </App>
  </HashRouter>
);
render(routes, ickt);
```

React声明组件有哪几种方法，有什么不同?

React 声明组件的三种方式：

- 函数式定义的 无状态组件
- ES5原生方式 `React.createClass` 定义的组件
- ES6形式的 `extends React.Component` 定义的组件

（1）无状态函数式组件 它是为了创建纯展示组件，这种组件只负责根据传入的props来展示，不涉及到state状态的操作 组件不会被实例化，整体渲染性能得到提升，不能访问this对象，不能访问生命周期的方法

（2）ES5 原生方式 React.createClass // RFC React.createClass会自绑定函数方法，导致不必要的性能开销，增加代码过时的可能性。

（3）E6继承形式 React.Component // RCC 目前极为推荐的创建有状态组件的方式，最终会取代React.createClass形式；相对于 React.createClass可以更好实现代码复用。

无状态组件相对于于后者的区别： 与无状态组件相比，React.createClass和React.Component都是创建有状态的组件，这些组件是要被实例化的，并且可以访问组件的生命周期方法。

React.createClass与React.Component区别：

① 函数this自绑定

- React.createClass创建的组件，其每一个成员函数的this都有React自动绑定，函数中的this会被正确设置。
- React.Component创建的组件，其成员函数不会自动绑定this，需要开发者手动绑定，否则this不能获取当前组件实例对象。

② 组件属性类型propTypes及其默认props属性defaultProps配置不同

- React.createClass在创建组件时，有关组件props的属性类型及组件默认的属性会作为组件实例的属性来配置，其中defaultProps是使用getDefaultProps的方法来获取默认组件属性的
- React.Component在创建组件时配置这两个对应信息时，他们是作为组件类的属性，不是组件实例的属性，也就是所谓的类的静态属性来配置的。

③ 组件初始状态state的配置不同

- React.createClass创建的组件，其状态state是通过getInitialState方法来配置组件相关的状态；
- React.Component创建的组件，其状态state是在constructor中像初始化组件属性一样声明的。

React中refs的作用是什么？有哪些应用场景？

Refs 提供了一种方式，用于访问在 render 方法中创建的 React 元素或 DOM 节点。Refs 应该谨慎使用，如下场景使用 Refs 比较适合：

- 处理焦点、文本选择或者媒体的控制
- 触发必要的动画
- 集成第三方 DOM 库

Refs 是使用 `React.createRef()` 方法创建的，他通过 `ref` 属性附加到 React 元素上。要在整个组件中使用 Refs，需要将 `ref` 在构造函数中分配给其实例属性：

javascript 复制代码

```javascript
class MyComponent extends React.Component {
  constructor(props) {
    super(props)
    this.myRef = React.createRef()
  }
  render() {
    return <div ref={this.myRef} />
  }
}
```

由于函数组件没有实例，因此不能在函数组件上直接使用 `ref`：

javascript 复制代码

```javascript
function MyFunctionalComponent() {
  return <input />;
}
class Parent extends React.Component {
  constructor(props) {
    super(props);
    this.textInput = React.createRef();
  }
  render() {
    // 这将不会工作!
    return (
      <MyFunctionalComponent ref={this.textInput} />
    );
  }
}
```

但可以通过闭合的帮助在函数组件内部进行使用 Refs：

javascript 复制代码

```javascript
function CustomTextInput(props) {
  // 这里必须声明 textInput，这样 ref 回调才可以引用它
  let textInput = null;
  function handleClick() {
    textInput.focus();
  }
  return (
    <div>
      <input
```

```
          type="text"
          ref={(input) => { textInput = input; }} />      <input
          type="button"
          value="Focus the text input"
          onClick={handleClick}
        />
      </div>
    );
  }
```

注意：

- 不应该过度的使用 Refs
- `ref` 的返回值取决于节点的类型：
  - 当 `ref` 属性被用于一个普通的 HTML 元素时，`React.createRef()` 将接收底层 DOM 元素作为他的 `current` 属性以创建 `ref` 。
  - 当 `ref` 属性被用于一个自定义的类组件时，`ref` 对象将接收该组件已挂载的实例作为他的 `current` 。
- 当在父组件中需要访问子组件中的 `ref` 时可使用传递 Refs 或回调 Refs。

参考 前端进阶面试题详细解答

React 组件中怎么做事件代理？它的原理是什么？

React基于Virtual DOM实现了一个SyntheticEvent层（合成事件层），定义的事件处理器会接收到一个合成事件对象的实例，它符合W3C标准，且与原生的浏览器事件拥有同样的接口，支持冒泡机制，所有的事件都自动绑定在最外层上。

在React底层，主要对合成事件做了两件事：

- **事件委派：** React会把所有的事件绑定到结构的最外层，使用统一的事件监听器，这个事件监听器上维持了一个映射来保存所有组件内部事件监听和处理函数。
- **自动绑定：** React组件中，每个方法的上下文都会指向该组件的实例，即自动绑定this为当前组件。

state 和 props 触发更新的生命周期分别有什么区别？

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()`

2. 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；
3. 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。

React 事件机制

```javascript 复制代码
<div onClick={this.handleClick.bind(this)}>点我</div>
```

React并不是将click事件绑定到了div的真实DOM上，而是在document处监听了所有的事件，当事件发生并且冒泡到document处的时候，React将事件内容封装并交由真正的处理函数运

行。这样的方式不仅仅减少了内存的消耗，还能在组件挂在销毁时统一订阅和移除事件。

除此之外，冒泡到document上的事件也不是原生的浏览器事件，而是由react自己实现的合成事件（SyntheticEvent）。因此如果不想要是事件冒泡的话应该调用event.preventDefault()方法，而不是调用event.stopProppagation()方法。 JSX 上写的事件并没有绑定在对应的真实 DOM 上，而是通过事件代理的方式，将所有的事件都统一绑定在了 `document` 上。这样的方式不仅减少了内存消耗，还能在组件挂载销毁时统一订阅和移除事件。

另外冒泡到 `document` 上的事件也不是原生浏览器事件，而是 React 自己实现的合成事件（SyntheticEvent）。因此我们如果不想要事件冒泡的话，调用 `event.stopPropagation` 是无效的，而应该调用 `event.preventDefault` 。

实现合成事件的目的如下：

- 合成事件首先抹平了浏览器之间的兼容问题，另外这是一个跨浏览器原生事件包装器，赋予了跨浏览器开发的能力；
- 对于原生浏览器事件来说，浏览器会给监听器创建一个事件对象。如果你有很多的事件监听，那么就需要分配很多的事件对象，造成高额的内存分配问题。但是对于合成事件来说，有一个事件池专门来管理它们的创建和销毁，当事件需要被使用时，就会从池子中复用对象，事件回调结束后，就会销毁事件对象上的属性，从而便于下次复用事件对象。

对React-Fiber的理解，它解决了什么问题?

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。**

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。

React中有使用过getDefaultProps吗？它有什么作用?

通过实现组件的getDefaultProps，对属性设置默认值（ES5的写法）：

javascript 复制代码

```
var ShowTitle = React.createClass({
  getDefaultProps:function(){
    return{
      title : "React"
    }
  },
  render : function(){
    return <h1>{this.props.title}</h1>
  }
});
```

React中的props为什么是只读的?

`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

在React中如何避免不必要的render?

React 基于虚拟 DOM 和高效 Diff 算法的完美配合，实现了对 DOM 最小粒度的更新。大多数情况下，React 对 DOM 的渲染效率足以业务日常。但在个别复杂业务场景下，性能问题依然

会困扰我们。此时需要采取一些措施来提升运行性能，其很重要的一个方向，就是避免不必要的渲染（Render）。这里提下优化的点：

- **shouldComponentUpdate 和 PureComponent**

在 React 类组件中，可以利用 shouldComponentUpdate或者 PureComponent 来减少因父组件更新而触发子组件的 render，从而达到目的。shouldComponentUpdate 来决定是否组件是否重新渲染，如果不希望组件重新渲染，返回 false 即可。

- **利用高阶组件**

在函数组件中，并没有 shouldComponentUpdate 这个生命周期，可以利用高阶组件，封装一个类似 PureComponet 的功能

- **使用 React.memo**

React.memo 是 React 16.6 新的一个 API，用来缓存组件的渲染，避免不必要的更新，其实也是一个高阶组件，与 PureComponent 十分类似，但不同的是， React.memo只能用于函数组件。

React setState 调用之后发生了什么？是同步还是异步？

（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；

React-Router如何获取URL的参数和历史对象?

（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

```javascript 复制代码
import { useHistory } from "react-router-dom";
let history = useHistory();
```

2.使用this.props.history获取历史对象

```javascript 复制代码
let history = this.props.history;
```

React-Router的实现原理是什么?

客户端路由实现的思想：

- 基于 hash 的路由：通过监听 `hashchange` 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

在React中组件的this.state和setState有什么区别?

this.state通常是用来初始化state的，this.setState是用来修改state值的。如果初始化了state之后再使用this.state，之前的state会被覆盖掉，如果使用this.setState，只会替换掉相应的state值。所以，如果想要修改state的值，就需要使用setState，而不能直接修改state，直接修改state之后页面是不会更新的。

React中可以在render访问refs吗？为什么？

javascript 复制代码

```
<>
  <span id="name" ref={this.spanRef}>{this.state.title}</span>
  <span>{     this.spanRef.current ? '有值' : '无值'  }</span>
</>
```

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：

React 高阶组件是什么，和普通组件有什么区别，适用什么场景

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。

高阶组件（HOC）就是一个函数，且该函数接受一个组件作为参数，并返回一个新的组件，它只是一种组件的设计模式，这种设计模式是由react自身的组合性质必然产生的。我们将它们称为纯组件，因为它们可以接受任何动态提供的子组件，但它们不会修改或复制其输入组件中的任何行为。

javascript 复制代码

```
// 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 <WrappedComponent data={this.state.data} {...this.props} />;
    }
  };

// 使用
const BlogPostWithSubscription = withSubscription(BlogPost,
  (DataSource, props) => DataSource.getBlogPost(props.id));
```

1）HOC的优缺点

- 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）适用场景

- 代码复用，逻辑抽象
- 渲染劫持
- State 抽象和更改
- Props 更改

3）具体应用例子

- **权限控制：** 利用高阶组件的 **条件渲染** 特性可以对页面进行权限控制，权限控制一般分为两个维度：页面级别和 页面元素级别

javascript 复制代码

```
// HOC.js
function withAdminAuth(WrappedComponent) {
    return class extends React.Component {
        state = {
            isAdmin: false,
        }
        async UNSAFE_componentWillMount() {
            const currentRole = await getCurrentUserRole();
            this.setState({
                isAdmin: currentRole === 'Admin',
            });
        }
        render() {
            if (this.state.isAdmin) {
                return <WrappedComponent {...this.props} />;
            } else {
                return (<div>您没有权限查看该页面，请联系管理员！</div>);
```

```
            }
        }
    };
}

// pages/page-a.js
class PageA extends React.Component {
    constructor(props) {
        super(props);
        // something here...
    }
    UNSAFE_componentWillMount() {
        // fetching data
    }
    render() {
        // render page with data
    }
}
export default withAdminAuth(PageA);


// pages/page-b.js
class PageB extends React.Component {
    constructor(props) {
        super(props);
    // something here...
        }
    UNSAFE_componentWillMount() {
    // fetching data
    }
    render() {
    // render page with data
    }
}
export default withAdminAuth(PageB);
```

- **组件渲染性能追踪：** 借助父组件子组件生命周期规则捕获子组件的生命周期，可以方便的对某个组件的渲染时间进行记录：

javascript 复制代码

```
class Home extends React.Component {
        render() {
            return (<h1>Hello World.</h1>);
        }
    }
    function withTiming(WrappedComponent) {
        return class extends WrappedComponent {
            constructor(props) {
```

```
            super(props);
            this.start = 0;
            this.end = 0;
        }
        UNSAFE_componentWillMount() {
            super.componentWillMount && super.componentWillMount();
            this.start = Date.now();
        }
        componentDidMount() {
            super.componentDidMount && super.componentDidMount();
            this.end = Date.now();
            console.log(`${WrappedComponent.name} 组件渲染时间为 ${this.end - this.start} ms`);
        }
        render() {
            return super.render();
        }
    };
}

export default withTiming(Home);
```

注意：withTiming 是利用 反向继承 实现的一个高阶组件，功能是计算被包裹组件（这里是 Home 组件）的渲染时间。

- **页面复用**

javascript 复制代码

```
const withFetching = fetching => WrappedComponent => {
    return class extends React.Component {
        state = {
            data: [],
        }
        async UNSAFE_componentWillMount() {
            const data = await fetching();
            this.setState({
                data,
            });
        }
        render() {
            return <WrappedComponent data={this.state.data} {...this.props} />;
        }
    }
}

// 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);
```

React Hooks 解决了哪些问题?

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