2024前端高频—React万字总结篇

1、说说对React的理解，有哪些特征？

理解：

- react是构建用户界面的JavaScript库，只提供了UI层面的解决方案。
- 遵循组件设计模式、声明式编程、函数式编程的概念。
- 使用虚拟DOM来操控实际DOM，遵循`高阶组件`到`低阶组件`的`单向数据流`。
- react可将界面形成独立的小块，每一个小块就是一个组件，这些组件之间可通过组合，嵌套构成整个界面。

特征：

- JSX语法
- 单项数据流
- 虚拟DOM
- 声明式编程
- 组件化

2、state和props有什么区别？

一个组件的形态可以由内部的数据状态及外部的参数所决定的，内部的数据状态：state，外部参数：props

state：一般存在于constructor，需要通过setState来进行修改。

props：父级所传递的参数，props一般在子组件内是不能够被修改的，只能通过外部传入进行修改。

区别：

相同点：

- 两者都是JavaScript对象
- 两者都用于保存数据信息

- state与props的更改都能引起render的重新执行，进行页面重新渲染。

不同点：

- state是多变的，可以修改的；props在组件内部是不可以被修改的。
- props是外部传入的，state是组件内部自己管理的。

3、super()和super(props)有什么区别？

4、说说对React中类组件和函数组件的理解？

类组件：

通过ES6的编写形式来编写组件，该类必须继承 `React.Cpmponent` 。通过 `this.props` 访问父组件传递的参数。在组件中需要有 `render` 方法，在return中返回React对象。

函数式组件：

通过编写函数的形式编写组件，函数的第一个参数为props，用于接受父组件参数。

区别：

- 编写形式的不同

```
class Welcome extends React.Component {
    constructor(props) {
        super(props)
    }
    render() {
        return <h1> Hello, {this.props.name}</h1>
    }
 }


const Welcome = (props) => {
    return <h1>Hello, {props.name}</h1>
```

```
}
```

- 状态管理的不同
  在Hooks出来之前，函数组件为无状态组件，不能够管理组件状态，而类组件可以通过state来管理，通过setState来进行修改。在Hooks出来之后，函数组件可以通过`useStete`来管理状态。
- 生命周期的不同
  函数式组件不存在生命周期，因为生命周期钩子函数都继承于`React.Cpmponent`。若需要使用生命周期，则改为使用类组件。但函数式组件可以通过`useEffect`来模拟生命周期的作用：

```
useEffect(() => {

}, []);


useEffect(() => {

}, [variable1, variable2]);

useEffect(() => {


  return () => {

  };
}, []);

useEffect(() => {


  if () {

  } else {
```

```
  }
}, [variable1, variable2]);
```

- 调用方式的不同
  函数式组件调用即执行函数即可，类组件需要对组件进行实例化，调用实例的render方法。

5、说说对受控组件和非受控组件的理解？场景？

受控组件：

接受控制的组件，组件的状态全程相应外部数据的传入。

非受控组件：

不接受控制的组件，一般情况下，只有在初始化的时候接受外部数据，再储存为自身状态。

应用场景：

受控组件：表单输入、复选框、单选框等需要实时更新UI的组件，以及需要对用户输入进行验证和处理的场景。

非受控组件：需要直接操作DOM元素的场景，比如使用第三方库或原生JavaScript来处理用户输入的组件，或者一些不需要实时更新UI的组件。

6、说说React的事件机制？

在React中，基于浏览器事件有一套自身的事件机制，包括：事件注册、事件合成、事件派发等，这些事件被称为合成事件。它的所有事件都是挂载在Document对象上，当真实DOM触发时，会冒泡到document上后，再处理react事件，所以会先执行原生事件，再处理react事件，最后再真正执行document上挂载事件。

7、Reactt事件绑定的方式有哪些？区别？

render方法中使用bind

```
class App extends React.Component {
    handleClick() {
        console.log('this > ', this);
    }
    render() {
        return (
            <div onClick={this.handleClick.bind(this)}>test</
div>
        )
      }
 }
```

render方法中使用箭头函数

```
class App extends React.Component {
    handleClick() {
        console.log('this > ', this);
    }
    render() {
        return (
            <div onclick={e => this.handleClick(e)}}>test</di
v>
            )
        }
}
```

constructor中使用bind

```
class App extends React.Component {
    constructor(props) {
        super(props);
        this.handleClick = this.handleClick.bind(this);
    }
    handleClick() {
        console.log('this > ', this);
    }
    render() {
        return (
            <div onclick={this.handClick}>test</div>
        )
    }
}
```

定义阶段使用箭头函数

```
class App extends React.Component {
    constructor(props) {
        super(props);
    }
    const handleClick = () => {
        console.log('this > ', this);
    }
    render() {
        return (
            <div onclick={this.handClick}>test</div>
        )
    }
}
```

区别:

- 编写方面:方式一、方式二写法简单，方式三的编写过于冗杂
- 性能方面:方式一和方式二在每次组件render的时候都会生成新的方法实例，性能问题欠缺。若该函数作为属性值传给子组件的时候，都会导致额外的渲染。而方式三、方式四只会生成一个方法实例。

综合上述，方式四是最优的事件绑定方式

8、React构建组件的方式有哪些？区别？

目前构建组件的方式有以下三种：

- 函数式创建

```
const HelloComponent =(props) => {
    return <div>Hello {props.name}</div>
}
```

- 通过React.createClass方法创建

```
function HelloComponent(props) {
return React.createElement(
    "div",
    null,
    "Hello ",
    props.name
    );
}
```

- 继承React.Component创建

```
class Timer extends React.Component {
    constructor(props) {
        super(props);
```

```
        this.state = { seconds: 0 };
    }
    tick() {
        this.setState(state => ({
            seconds: state.seconds + 1
        }));
    }
    componentDidMount() {
        this.interval = setInterval(() => this.tick(), 1000);
    }
    componentWillUnmount() {
        clearInterval(this.interval);
    }
    render() {
        return (
            <div>Seconds: {this.state.seconds}</div>
        );
    }
}
```

由于 `React.createClass` 创建的方式过于冗杂，并不建议使用。

而像函数式创建和类组件创建的区别主要在于需要创建的组件是否需要为有状态组件:

- 对于一些无状态的组件创建，建议使用函数式创建的方式
- 由于 react hooks 的出现，函数式组件创建的组件通过使用 hooks 方法也能使之成为有状态组件，再加上目前推崇函数式编程，所以这里建议都使用函数式的方式来创建组件

在考虑组件的选择原则上，能用无状态组件则用无状态组件

9、说说React中引入css的方式有哪几种？区别？

常见的引入方式:

- 在组件内直接使用

- 组件中引入.css文件
- 组件中引入.modules.css文件
- CSS in JS

区别：

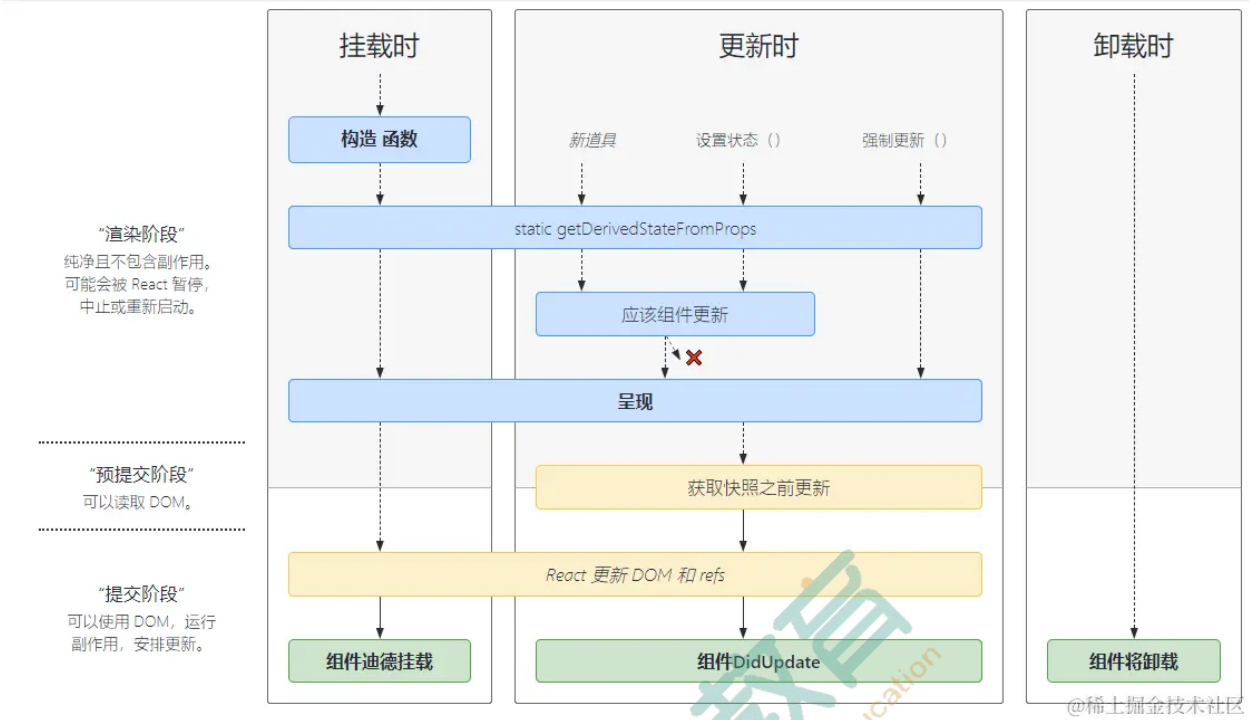
- 在组件内直接使用 css 该方式编写方便，容易能够根据状态修改样式属性，但是大量的演示编写容易导致代码混乱。
- 组件中引入 .css 文件符合我们日常的编写习惯，但是作用域是全局的，样式之间会层叠。
- 引入.module.css 文件能够解决局部作用域问题，但是不方便动态修改样式，需要使用内联的方式进行样式的编写。
- 通过css injs 这种方法，可以满足大部分场景的应用，可以类似于预处理器一样样式嵌套、定义、修改状态等。

至于使用 react 用哪种方案引入css，并没有一个绝对的答案，可以根据各自情况选择合适的方案。

10、说说React生命周期有哪些不同阶段？

生命周期主要分为三个阶段：

- 创建阶段: `constructor`、`getDerivedStateFromProps`、`render`、`componentDidMount`
- 更新阶段：`getDerivedStateFromProps`、`shouldComponentUpdate`、`render`、`getSnapshotBeforeUpdate`、`componentDidUpdate`
- 卸载阶段：`componentWillUnmount`



11、React中组件之间是如何通信的？

react中通信方式有以下几种：

- `父向子`：父组件在调用子组件的时候，只需要在子组件标签内传递参数，子组件通过props属性就能接收父组件传递过来的参数。
- `子向父`：子组件向父组件通信的基本思路是，父组件向子组件传一个函数，然后通过这个函数的回调，拿到子组件传过来的值。
- `兄弟间`:父组件作为中间层来实现数据的互通，通过使用父组件传递。
- `父向后代`：使用 context 提供了组件之间通讯的一种方式，可以共享数据，其他数据都能读取对应的数据。
- `非关系`：使用redux，创建全局资源数据管理，共用实现通信。

12、说说对高阶组件的理解？应用场景？

高阶组件的特点：1.`接受一个或多个函数作为参数输入` 2.`返回输出一个函数`。

JavaScript中常见的高阶函数有：`map`、`filter`、`reduce`、`forEach`、`sort`、`find`、`some`、`every`、`flatMap`、`reduceRight`、`findIndex`、`indexOf`、`lastIndexOf`等

react中常见的高阶函数有：`withRouter`、`connect`、`memo`、`forwardRef`、`Suspense`等。

在 React 中使用高阶函数的场景包括：

1. **代码复用：** 当多个组件需要相同的逻辑时，可以将这部分逻辑提取到一个高阶函数中，然后将这个函数应用到需要的组件上，以实现代码复用。
2. **逻辑抽象：** 高阶函数可以将通用逻辑从组件中抽象出来，使组件更加专注于展示数据或处理用户交互。
3. **性能优化：** 使用 memo 高阶函数可以优化函数组件的性能，避免不必要的重新渲染。
4. **状态管理：** 使用高阶函数连接 Redux 或者其他状态管理库，可以将状态管理逻辑与组件分离，使组件更加专注于 UI 展示。

13、在React中组件间过渡动画如何实现？

14、说说你在React项目中是如何捕获错误的？

组件级别的错误处理

在组件内部，我们可以通过try-catch块来捕获错误。例如，当我们执行某些可能会抛出错误的操作时：

```
class MyComponent extends React.Component {
  componentDidMount() {
    try {

    } catch (error) {
      console.error(error);

    }
  }
}
```

使用错误处理库

可以使用诸如Sentry这样的错误处理库来自动捕获和记录错误。这些库通常提供了丰富的配置选项和集成工具，可以帮助开发者快速集成错误报告功能。

React的错误边界（Error Boundaries）

React 16引入了错误边界的概念，它允许我们通过高阶组件来捕获并处理子组件树中发生的任何错误。

```
class ErrorBoundary extends React.Component {
  constructor(props) {
    super(props);
    this.state = { hasError: false };
  }

  static getDerivedStateFromError(error) {

    return { hasError: true };
  }

  componentDidCatch(error, errorInfo) {

    console.error(error, errorInfo);
  }

  render() {
    if (this.state.hasError) {

      return <h1>Something went wrong.</h1>;
    }

    return this.props.children;
  }
}
```

在上面的示例中，ErrorBoundary 组件可以在其子组件树中的任何位置捕获错误。当捕获到错误时，getDerivedStateFromError 生命周期方法会被调用，我们可以在这里设置

组件的状态以渲染错误UI。componentDidCatch 生命周期方法则可以用来记录错误或执行其他错误处理操作。 除此之外，还可以使用监听onerror事件：

```
window.addEventListener('error', function(event) { ... })
```

错误边界的限制

虽然错误边界是一个非常有用的特性，但它并不能捕获所有类型的错误。以下是一些错误边界无法捕获的情况：

- 事件处理器中的错误（例如，点击事件）
- 异步代码（例如，setTimeout或requestAnimationFrame回调函数）
- 服务端渲染过程中发生的错误
- 错误边界自身抛掷的错误（不会被同一个错误边界捕获）
- 因此，除了使用错误边界之外，我们还需要结合其他错误处理策略来确保应用程序的健壮性。

15、说说对React的refs的理解？应用场景？

创建refs有以下几种方式：

1、传入字符串，使用时通过this.refs传入的字符串的格式获取对应的元素

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

```
this.refs.myref.innerHTML = "hello";
```

2、 传入对象，对象是通过 React.createRef() 方式创建出来，使用时获取到创建的对象中存在 current 属性就是对应的元素。

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



const node = this.myRef.current;
```

3、 传入函数，当`ref`传入为一个函数的时候，在渲染过程中，回调函数参数会传入一个元素对象，然后通过实例将对象进行保存。

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

```
const node = this.myref
```

4、 传入hook，hook是使用useRef()方式创建

```
function App(props) {
    const myref = useRef()
    return (
        <div ref={myref} />
    )
}


const node = myref.current
```

使用场景：

- 对Dom元素的焦点控制、内容选择、控制。
- 对Dom元素的内容设置及媒体播放
- 对Dom元素的操作和对组件实例的操作
- 集成第三方 DOM 库

16、说说React中的setState执行机制？

在使用setState更新数据时，更新类型分为 异步更新 、 同步更新

异步更新

在react合成事件内同步执行的setState是异步的，如onClick等

- **React控制的事件处理：** 当setState在React的生命周期方法或事件处理函数中被调用时，React会将多个setState调用批处理为一个更新，它们会在事件处理结束后一

起被应用，并且可能会导致只有一次重新渲染。

- **React Hook：** 使用`useState` hook中的更新函数时，同样也是React控制调度的，与setState有类似的表现。

同步更新

在宏任务：`setTimeout` ，微任务：`.then` ，或直接在DOM元素上绑定的事件内是同步的。

- **setTimeout 或 setInterval 回调**：当在这些函数中调用`setState()`时，状态更新和重渲染通常会在调用它们的下一行代码之前完成。
- **原生事件处理**：如果你在React组件中绑定了原生事件（使用 addEventListener 方式）并在事件回调中调用`setState()`，React无法控制该事件处理中的状态更新，因此状态更新通常会同步发生。
- **异步代码**：在Promise、async/await、或者其他异步API的回调中调用`setState()`，会导致立即更新状态并重新渲染，因为React无法控制这些情况下的状态更新和调度。

17、说说React的render方法的原理？何时被触发？

原理：

在react中render的表现形式有两种：

在类组件中，render就为render方法：

```
class Foo extends React.Component {
    render() {
        return <p> TEXT </p>;
    }
}
```

在函数式组件中就为函数本身：

```
function Foo() {
    return return <p> TEXT </p>;
```

```
}
```

在render中，我们编写JSX，JSX通过babel编译后形成js：

触发条件：

- 类组件调用setState修改状态
- 函数式组件通过useState hook修改状态
- 类组件重新渲染
- 函数式组件重新渲染

18、说说Real DOM和Virtual DOM的区别？

Real DOM

Real DOM即为真实的DOM元素如div等。

Virtual DOM

Virtual DOM本质上是以JavaScript对象的形式对DOM的描述。在react中，JSX是其中一大特征，在js通过使用XML的方式去声明界面的DOM结构。

区别

- 虚拟 DOM 不会进行排版与重绘操作，而真实 DOM 会频繁重排与重绘
- 虚拟 DOM 的总损耗是“虚拟 DOM 增删改+真实 DOM 差异增删改+排版与重绘”，真实 DOM 的总损耗是“真实 DOM 完全增删改+排版与重绘

优缺点

Real DOM 优点：

- 易用

Real DOM 缺点：

- 效率低，解析速度慢，内存占用量过高性能差；
- 频繁操作真实 DOM，易于导致重绘与回流

Virtual DOM 优点：

- 简单方便:如果使用手动操作真实D0M来完成页面，繁琐又容易出错，在大规模应用下维护起来也很困难
- 性能方面:使用 Virtual DOM，能够有效避免真实 DOM 数频繁更新，减少多次引起重绘与回流提高性能
- 跨平台:React借助虚拟 DOM，带来了跨平台的能力，一套代码多端运行

Virtual DOM 缺点：

- 在一些性能要求极高的应用中虚拟 DOM 无法进行针对性的极致优化
- 首次渲染大量 DOM 时，由于多了一层虚拟 DOM 的计算，速度比正常稍慢

19、说说React的JSX转换成真实DOM的过程？

- 使用React.createElement或JSX编写React组件，实际上所有的JSX 代码最后都会转换成React.createElement(...)，Babel帮助我们完成了这个转换的过程。
- createElement函数对key和ref等特殊的props进行处理，并获取defaultProps对默认props进行赋值，并且对传入的孩子节点进行处理，最终构造成一个虚拟DOM对

象。

- ReactDOM.render将生成好的虚拟DOM渲染到指定容器上，其中采用了批处理、事务等机制并且对特定浏览器进行了性能优化，最终转换为真实DOM。

20、说说对Fiber架构的理解？解决了什么问题？

Fiber架构是React 16中引入的新的协调引擎，它重新实现了React的核心算法。这个协调引擎负责渲染界面和计算组件变化。Fiber架构的主要目标是增强React在渲染大型应用和执行动画、手势等交互动作时的性能表现。

理解Fiber架构

1. **任务可中断**: Fibe 架构的核心特性是其工作单元的'单位任务'可以中断和恢复。即React可以按需暂停渲染更新工作，去执行更高优先级的任务，然后再返回继续之前的渲染工作。
2. **增量渲染**: Fiber为React提供了增量渲染的能力，即将渲染任务拆分成一系列小的工作单元，每个工作单元完成时React可以暂停处理任务，检查是否有更高优先级的工作需要完成。
3. **任务优先级**: Fiber允许React根据任务的类型和上下文为不同的更新设置不同的优先级。例如，动画或用户交互的更新会比其他类型的状态更新有更高的优先级。
4. **更好的生命周期管理**: Fiber引入了新的生命周期方法，并更新了某些旧的生命周期方法来更好地适应异步渲染。

解决的问题

1. **避免阻塞**: 在老的React版本中，正在执行的更新过程不能被中断，这会导致应用在处理大量的UI更新时出现卡顿，影响用户交互。Fiber通过使渲染任务可中断并进行任务切片来解决此问题。
2. **改善动画和交互性能**: 由于可以中断渲染任务，Fiber可以确保关键的用户交互和动画在需要时可以得到及时处理，从而提供更流畅的用户体验。
3. **优先级调度**: 在大型应用中，有些更新操作比其他操作更为紧急，Fiber通过任务优先级确保更重要的更新可以优先执行，对比之前的处理方式，这大大改善了应用的响应性能。
4. **更稳健的错误处理**: Fiber架构引入了错误边界（Error Boundaries），它允许组件捕获并处理子组件树中的JavaScript错误，防止整个应用崩溃。

总而言之，Fiber架构让React变得更加强大和灵活，它通过任务的可中断性和优先级调度，使得React更好地适应复杂应用的渲染需求，提供了更流畅的用户体验，并允许React开发者有更细粒度的控制组件的渲染行为。

21、React中key值有什么作用？

react也存在Diff算法，key值的存在就是用于判断元素是创建还是移动，从而减少不必要的渲染。因此key的值需要为每一个元素赋予一个确定的值。良好的key属性是优化的非常关键的一步，使用时的注意事项为：

- key应该是唯一的
- key不能使用随机数，随机数会在每次render时，重新生成一个数字
- 不能使用index作为key值，对性能没有作用

22、说说React中diff的原理是什么？

react中diff算法遵循三个层级策略：

- tree层级
  DOM节点跨层级的操作不做优化，只会对相同层级的节点进行比较只有`创建、删除`操作，没有移动。
- component层级
  如果是相同类组件，则会继续向下diff运算，如果不是同一个类组件，那么直接删除这个组件下所有节点，创建新的。
- element层级
  对于比较同一层级的节点们，每个节点会在对应层级用唯一的key值作为唯一标识。
  element层级提供了 3 种节点操作：
  `INSERT_MARKUP（插入）`、`MOVE_EXISTING（移动）`、`REMOVE_NO DE（删除）`。

23、说说对React Hooks的理解?解决了什么问题?

Hook是react 16.8的新特征，可在不写class类组件的情况下，使用`state`以及其他react类组件的特征。解决了函数组件中无法使用状态和副作用的问题，让函数组件具有了更多类组件的特性，同时也让代码更加简洁、可读性更强、逻辑更加清晰。常见的hook如`useState`、`useEffect`等。

hooks能够更容易解决状态相关的重用问题：

- 每调用useHook一次都会生成一份独立的状态
- 通过自定义hook能够更好的封装我们的功能

编写 hooks 为函数式编程，每个功能都包裹在函数中，整体风格更清爽，更优雅hooks的出现，使函数组件的功能得到了扩充，拥有了类组件相似的功能，在我们日常使用中，使用hooks 能够解决大多数问题，并且还拥有代码复用机制。

24、说说你是如何提高组件的渲染效率的？在React中如何避免不必要的render？

提高渲染效率可以使用以下方式：

- **使用shouldComponentUpdate：** 确定是否可以跳过重新渲染；
- **使用PureComponent：** 当 props 和 state 与之前保持一致时会跳过重新渲染；
- 使**用React.memo或者useMemo来对组件进行缓存；**
- **优化state和props：**
  尽量减少组件的state和props，只保留必要的部分。对于大型对象或数组，尽量使用不可变数据结构，或者提供一个新的对象或数组，而不是直接修改原对象或数组。这样可以避免不必要的渲染，因为React会使用浅比较来检查props和state是否发生了变化;
- **使用列表的键（key）：**
  当渲染列表时，确保每个列表项都有一个唯一的key。这样，React就可以准确地识别哪些项发生了变化，哪些项没有变化，从而避免不必要的渲染。
- **使用 `React.lazy` 和 `Suspense` 懒加载组件：**
  如果你的应用有很多大型组件，或者有一些只在特定条件下才需要的组件，你可以使用
  `React.lazy` 和 `Suspense` 进行代码拆分。这样，只有当需要渲染这些组件时，才会加载它们的代码，从而减少了初始加载时间，并提高了渲染效率。
- **拆分组件：**
  将大型组件拆分为多个小型组件，可以提高渲染效率。 因为当父组件的状态或props发生变化时，只有与这些状态或props相关的子组件才会重新渲染，而其他子组件则不会受到影响。
- **使用context和hooks管理状态：**
  避免在组件树中通过props逐层传递状态，而是使用React的context和hooks来管理

状态。这样，你可以将状态存储在更高级别的组件中，并通过context和hooks在需要的地方访问它，从而减少了不必要的props传递和渲染。

- **避免使用内联函数：**
  使用内联函数，则每次调用render函数时都会创建一个新的函数实例。

25、说说React性能优化的手段有哪些？

提升组件渲染效率

参考上题

事件绑定的方式

在事件绑定方式中，我们了解到四种事件绑定的方式从性能方面考虑，在render方法中使用 bind 和render 方法中使用箭头函数这两种形式在每次组件 render 的时候都会生成新的方法实例，性能欠缺而constructor 中 bind 事件与定义阶段使用箭头函数绑定这两种形式只会生成一个方法实例，性能方面会有所改善。

避免使用 内敛对象

使用内联对象时，react会在每次渲染时重新创建对此对象的引用，这会导致接收此对象的组件将其视为不同的对象。因此，该组件对于props的千层比较始终返回false，导致组件一直渲染。

```
function Component(props) {
  const aProp = { someProp: 'someValue' }
  return <AComponent style={{ margin: 0 }} aProp={aProp} />
}


const styles = { margin: 0 };
function Component(props) {
  const aProp = { someProp: 'someValue' }
  return <AComponent style={styles} {...aProp} />
}
```

避免使用 匿名函数

虽然匿名函数是传递函数的好方法，但它们在每次渲染上都有不同的引用。类似于内联对象。为了保证作为props传递给react组件的函数的相同引用，如果使用的类组件可以将其声明为类方法，如果使用的函数组件，可以使用useCallback钩子来保持相同的引用。

```
function Component(props) {
  return <AComponent onChange={() => props.callback(props.i
d)} />
}


function Component(props) {
  const handleChange = useCallback(() => props.callback(prop
s.id), [props.id]);
  return <AComponent onChange={handleChange} />
}


class Component extends React.Component {
  handleChange = () => {
   this.props.callback(this.props.id)
  }
  render() {
    return <AComponent onChange={this.handleChange} />
  }
}
```

组件懒加载

使用`React.lazy`和`React.Suspense`完成延迟加载不是立即需要的组件。React加载的组件越少，加载组件的速度越快。

使用React.Fragment避免添加额外的DOM

用户创建新组件时，每个组件应具有单个父标签。父级不能有两个标签，所以顶部要有一个公共标签所以我们经常在组件顶部添加额外标签 div这个额外标签除了充当父标签之外，并没有其他作用，这时候则可以使用 fragement其不会向组件引入任何额外标记，但它可以作为父级标签的作用：

```
export default class NestedRoutingComponent extends React.Com
ponent {
    render() {
        return (
            <>
                <h1>This is the Header Component</ h1>
                <p>Welcome To Demo Page</ p>
            <>
        )
    }
}
```

服务器端渲染

采用服务端渲染端方式，可以使用户更快的看到渲染完成的页面服务端渲染，需要起一个node 服务，可以使用express、koa等，调用react的renderToString 方法，将根组件染成字符串，再输出到响应中。

26、说说你对React Router的理解？常用的router组件有哪些？

react-router 等前端路由的原理大致相同，可以实现无刷新的条件下切换显示不同的页面路由的本质就是页面的`URL`发生改变时，页面的显示结果可以根据`URL`的变化而变化，但是页面不会刷新，因此，可以通过前端路由可以实现单页(SPA)应用。react-router 主要分成了几个不同的包:

- `react-router:`实现了路由的核心功能
- `react-router-dom:`基于react-router，加入了在浏览器运行环境下的一些功能
- `react-router-native:`基于react-router，加入了react-native 运行环境下的一些功能

- `react-router-config:`用于配置静态路由的工具库

`react-router-dom`中常用的API，提供了一些组件，包括：

- BrowserRouter、HashRouter
  来区分项目路由模式为history模式还是hash模式
- Route
  用于路径的匹配，然后进行组件的染，对应的属性如下:
  1、path 属性:用于设置匹配到的路径
  2、component 属性:设置匹配到路径后，渲染的组件
  3、render 属性:设置匹配到路径后，渲染的内容
  4、exact 属性:开启精准匹配，只有精准匹配到完全一致的路径，才会渲染对应的组件
- Link、NavLink
  通常路径的跳转是使用 Link 组件，最终会被渲染成`a`元素，其中属性`to`代替`a`标题的`href`属性。
  NavLink 是在 Link 基础之上增加了一些样式属性，例如组件被选中时，发生样式变化，则可以设置 NavLink 的以下属性:
  1、`activeStyle:`活跃时(匹配时)的样式
  2、`activeClassName:`活跃时添加的class
- switch
  swich 组件的作用适用于当匹配到第一个组件的时候，后面的组件就不应该继续匹配。
- redirect
  用于路由的重定向，当这个组件出现时，就会自行跳转到对应的to路径中。

React中常用的几种路由跳转方式

push跳转

```
import { Link } from 'react-router-dom';
import { Link } from 'apollo/router'
```

```
<Link to="/detail">详情</Link>


<Link to={`/detail/${item.id}/${item.title}`}>详情</Link>
<Link to='/detail/01/详情1'>详情</Link>


this.props.history.push("/home/detail")


this.props.history.push("/detail",{id:"01",title:"详情"})
```

replace跳转

```
<Link replace to='/detail/'>详情</Link>


<Link replace to='/detail/01/详情1'>详情</Link>


this.props.history.replace("/detail")


this.props.history.replace("/detail",{id:"01",title:"详情1"})
```

goBackt跳转（回退）

```
this.props.history.goBack()
```

goForward跳转（前进）

```
this.props.history.goForward()
```

go跳转（向前/向后跳转，指定步数）

```
this.props.history.go(num)
```

27、说说React的Router有几种模式？实现原理？

在单页应用中，一个 web 项目只有一个 html 页面，一旦页面加载完成之后，就不用因为用户的操作而进行页面的重新加载或者跳转，其特性如下:

- 改变 url 且不让浏览器像服务器发送请求
- 在不刷新页面的前提下动态改变浏览器地址栏中的URL地址

其中主要分成了两种模式:

- hash 模式:在url后面加上#，如http://127.0.0.1:5500/home/#/page1[1]
- history 模式:允许操作浏览器的曾经在标签页或者框架里访问的会话历史记录。

实现原理百度吧，太吉尔长了。

28、你在React项目中是如何使用redux的？项目结构是如何划分的？

在react项目中，redux是用于数据状态管理，而react是一个视图层面的库，如果将两者连接在一起，可以使用官方推荐react-redux库，其具有高效且灵活的特性。

react-redux将组件分成：

- 容器组件：存在逻辑处理
- UI 组件：只负责现显示和交互，内部不处理逻辑，状态由外部控制

通过redux将整个应用状态存储到`store`中，组件可以派发`dispatch`行为`action`给`store`，其他组件通过订阅`store`中的状态`state`来更新自身的视图。

使用`react-redux`分成了两大核心： `Provider` `connection`

Provider

在`redux`中存在一个`store`用于存储`state`，如果将这个`store`存放在顶层元素中，其他组件都被包裹在顶层元素之上，那么所有的组件都能够受到`redux`的控制，都能够获取到`redux`中的数据。

```
<Provider store = {store}>
    <App />
<Provider>
```

connection

`connect`方法将`store`上的`getState`和`dispatch`包装成组件的`props`。

```
import { connect } from "react-redux";

connect(mapStateToProps, mapDispatchToProps)(MyComponent)
```

29、说说你对redux中间件的理解？常用的中间件有哪些？实现原理？

中间件(Middleware)是介于应用系统和系统软件之间的一类软件，它使用系统软件所提供的基础服务(功能)，衔接网络上应用系统的各个部分或不同的应用，能够达到资源共享、功能共享的目的那么如果需要支持异步操作，或者支持错误处理、日志监控，这个过程就可以用上中间件Redux 中，中间件就是放在就是在`dispatch`过程，在分发`action`进行拦截处理。其本质上一个函数，对`store.dispatch`方法进行了改造，在发出`Action`和执行`Reducer`这两步之间，添加了其他功能。

有很多优秀的 redux 中间件，如:

- redux-thunk：用于异步操作。
- redux-logger：用于日志记录

30、说说你对immutable的理解？如何应用在React项目中？

什么是Immutable

Immutable，不可改变的，在计算机中，即指一旦创建，就不能再被更改的数据对 Immutable 对象的任何修改或添加制除操作都会返回一个新的 Immutable 对象 Immutable 实现的原理是 Persistent Data Structure(持久化数据结构):

- 用一种数据结构来保存数据；
- 当数据被修改时，会返回一个对象，但是新的对象会尽可能的利用之前的数据结构而不会对内存造成浪费；也就是使用旧数据创建新数据时，要保证旧数据同时可用且不变，同时为了避免deepCopy 把所有节点都复制一遍带来的性能损耗，Immutable 使用了 Structural Sharing (结构共享)如果对象树中一个节点发生变化，只修改这个节点和受它影响的父节点，其它节点则进行共享。

使用

使用 Immutable 对象最主要的库是immutable.jsimmutable.js 是一个完全独立的库，无论基于什么框架都可以用它其出现场景在于弥补 Javascript 没有不可变数据结构的问题，通过 structural sharing来解决的性能问题内部提供了一套完整的 Persistent Data Structure，还有很多易用的数据类型，如Collection、List、Map、Set、Record、Seq，其中:

- List; 有序索引集，类似 JavaScript 中的 Array
- Map: 无序索引集，类似 JavaScript 中的 Object
- Set: 没有重复值的集合

主要的方法如下:

- fromJS():将一个js数据转换为Immutable类型的数据

```
const obj=Immutable.fromJS({a:'123',b:'234'})
```

- toJS():将一个Immutable数据转换为JS类型的数据
- is():对两个对象进行比较

```
import { Map,is }from 'immutable'
const map1=Map({a:1,b:1,c:1 })
const map2=Map({a:1，b:1，c:1 })
map1 === map2
0bject.is(map1,map2)
is(map1,map2)
```

- get(key):对数据或对象取值
- getIn([]):对嵌套对象或数组取值，传参为数组，表示位置

```
Let abs =Immutable.fromJs({a{b:2}});
abs.getIn(['a',，'b'])
abs.getIn(['a'，'c'])
let arr = Immutable.fromJS([1,2,3,{a: 5}]);
arr.getIn([3，'a']);
arr.getIn([3，'c']);
```

如下例子：

```
import Immutable from "immutable";

foo = Immutable.fromJsS({a:{b:1}});

bar = foo.setIn(['a'，'b'],2);
console.log(foo.getIn(['a','b']));
console.log(foo === bar);
```

如果换到原生的 js ，则对应如下:

```
let foo =fa:{b: 1}};
let bar = foo;
```

```
bar.a.b = 2;
console.log(foo.a.b);
console.log(foo === bar);
```

31、React组件复用方式有哪些？

为什么要进行组件复用？

在 React 开发中，组件逻辑复用对于提升代码可维护性、促进代码共享以及简化复杂应用程序的构建至关重要。通过复用代码，可以避免重复，让代码库更加精简和可管理。

有哪些？

高阶组件（HOC）

HOC 将一个组件包装在另一个组件中，允许增强或修改组件的行为。这是一种提取公共逻辑并将其抽象到独立组件中的强大技术。

```
const withLoading = (Component) => (props) => {
    const [isLoading, setIsLoading] = useState(true);
    useEffect(() => {

        setTimeout(() => {
            setIsLoading(false);
        }, 1000);
    }, []);
    return (
        <Component {...props} isLoading={isLoading} />
    );
};
```

Render Props

Render props 是一种将渲染函数作为 props 传递给子组件的技术。子组件可以使用该函数渲染其自身。这允许高度可定制的逻辑复用。

```
const CounterContext = createContext();

const CounterProvider = ({ children }) => {
    const [count, setCount] = useState(0);
        return (
            <CounterContext.Provider value={{ count, setCount
}}>
                {children}
            </CounterContext.Provider>
        );
};
 const useCounter = () => {
     const context = useContext(CounterContext);
     return [context.count, context.setCount];
 };
```

自定义 Hooks

自定义 Hooks 允许在组件外部定义和复用状态逻辑。通过自定义 Hooks，可以将共享逻辑封装到独立函数中，并在多个组件中使用。

```
const useTitle = (title) => {
    useEffect(() => {
        document.title = title;
    }, [title]);
};
```

Context API

Context API 允许组件在整个组件树中共享数据，而无需显式传递 props。这对于管理共享状态和减少 props 嵌套非常有用。

```
const MyContext = createContext(null);

const MyProvider = ({ children }) => {
    const [state, setState] = useState(0);
    return (
        <MyContext.Provider value={{ state, setState }}>
            {children}
        </MyContext.Provider>
    );
};
const useMyContext = () => {
    const context = useContext(MyContext);
    return context.state;
};
```

32、React生命周期有哪些？React16废弃了哪些？为什么要废弃？新增的生命周期钩子有哪些？有什么作用？

生命周期参考第 10 题。

被废弃的三个函数都是在render之前，因为Fiber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`componentWilIMount`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。

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Fiber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。

33、React的setState是同步的还是异步的？

参考 16 题。

34、讲讲React的hooks，有什么好处？有哪些常用的hooks？

是什么？及好处。

React Hooks 是 React 16.8 版本引入的一项重要特性，它允许在函数组件中使用状态(state)和其他 React 特性，而无需编写 class 组件。Hooks 提供了一种更简洁、更灵活的方式来编写组件，具有以下几个好处：

- 更简洁的代码：
  相比于使用 class 组件，使用 Hooks 可以更轻松地管理组件的状态和副作用，代码更加简洁清晰。
- 逻辑复用：
  Hooks 可以将组件的状态逻辑抽象为可复用的函数，使得逻辑可以在多个组件中共享，提高了代码的可维护性和复用性。
- 减少代码量：
  相比于使用 class 组件时需要编写大量的模板代码（如 constructor、render 方法等），使用 Hooks 可以减少很多模板代码，
- 代码更加精简。
  更好的性能优化：Hooks 可以在不引入额外的组件层级的情况下管理状态和副作用，从而更好地进行性能优化。

常用Hooks

useState

useState用于在函数组件中管理状态。它返回一个包含当前状态和一个更新状态的函数的数组。更新状态的函数可以接受一个新的值，并更新状态。

```
import React, { useState } from 'react';

function Counter() {
  const [count, setCount] = useState(0);

  function handleClick() {
    setCount(count + 1);
  }

  return <button onClick={handleClick}>Count: {count}</button
>;
```

```
}
```

useEffect

useEffect用于在函数组件中处理副作用。它接受两个参数：一个副作用函数和一个依赖数组。当依赖数组中的任何一个值发生变化时，副作用函数将被调用。

```
import React, { useState, useEffect } from 'react';

function Timer() {
  const [time, setTime] = useState(0);

  useEffect(() => {
    const interval = setInterval(() => {
      setTime(time + 1);
    }, 1000);

    return () => clearInterval(interval);
  }, [time]);

  return <div>Time: {time}</div>;
}
```

useContext

useContext用于在函数组件中使用上下文。它接受一个上下文对象并返回上下文的当前值。当上下文的值发生变化时，函数组件将重新渲染。

```
import React, { createContext, useContext } from 'react';

const ThemeContext = createContext('light');

function Header() {
  const theme = useContext(ThemeContext);
```

```
  return (
    <header style={{ background: theme }}>
      <h1>My App</h1>
    </header>
  );
}

function App() {
  return (
    <ThemeContext.Provider value="dark">
      <Header />
    </ThemeContext.Provider>
  );
}
```

useReducer

useReducer用于在函数组件中管理复杂的状态。它接受一个reducer函数和一个初始状态，并返回一个包含当前状态和一个派发操作的数组。派发操作将一个操作对象发送到reducer函数中，并返回一个新的状态。

```
import React, { useReducer } from 'react';

const initialState = { count: 0 };

function reducer(state, action) {
  switch (action.type) {
    case 'increment':
      return { count: state.count + 1 };
    case 'decrement':
      return { count: state.count - 1 };
    default:
      throw new Error();
```

```
  }
}

function Counter() {
  const [state, dispatch] = useReducer(reducer, initialStat
e);

  function handleIncrement() {
    dispatch({ type: 'increment' });
  }

  function handleDecrement() {
    dispatch({ type: 'decrement' });
  }

  return (
    <>
      <button onClick={handleIncrement}>+</button>
      <span>{state.count}</span>
      <button onClick={handleDecrement}>-</button>
    </>
  );
}
```

useCallback

useCallback用于在函数组件中缓存回调函数。 它接受一个回调函数和一个依赖数组，并返回一个缓存的回调函数。当依赖数组中的任何一个值发生变化时，缓存的回调函数将被更新。

```
import React, { useState, useCallback } from 'react';

function Parent() {
  const [count, setCount] = useState(0);
```

```
  const handleClick = useCallback(() => {
    setCount(count + 1);
  }, [count]);

  return (
    <>
      <Child onClick={handleClick} />
      <span>Count: {count}</span>
    </>
  );
}

function Child({ onClick }) {
  return <button onClick={onClick}>Click me</button>;
}
```

useMemo

useMemo用于在函数组件中缓存计算结果。它接受一个计算函数和一个依赖数组，并返回一个缓存的计算结果。当依赖数组中的任何一个值发生变化时，计算函数将被重新计算。

```
import React, { useMemo } from 'react';

function ExpensiveComponent({ value1, value2 }) {
  const result = useMemo(() => {
    console.log('calculating result');
       return value1 * value2;
  }, [value1, value2]);

  return <div>Result: {result}</div>;
}
```

useRef

useRef用于在函数组件中引用DOM元素或其他值。它返回一个包含可变引用的对象。当在函数组件中传递该对象时，它将保留其引用，即使组件重新渲染。

```
import React, { useRef } from 'react';

function Input() {
  const inputRef = useRef(null);

  function handleClick() {
    inputRef.current.focus();
  }

  return (
    <>
      <input ref={inputRef} />
      <button onClick={handleClick}>Focus input</button>
    </>
  );
}
```

汇总

`useState：`用于在函数组件中管理状态。可以用于跟踪和更新组件的内部状态，例如表单输入、开关状态、计数器等。

`useEffect：`用于处理副作用操作，例如数据获取、订阅事件、DOM操作等。可以在组件渲染后执行一些操作，也可以在组件卸载前进行清理操作。

`useContext：`用于在组件之间共享数据。可以创建一个全局的上下文，并在组件树中的多个组件中访问和更新该上下文。

`useReducer：`用于管理复杂的状态逻辑。可以用于替代useState，特别适用于具有复杂状态转换的组件，例如有限状态机、游戏状态等。

`useCallback：`用于性能优化。可以缓存函数实例，以便在依赖项不变的情况下避免不必要的函数重新创建，提高组件的性能。

`useMemo：`用于性能优化。可以缓存计算结果，以便在依赖项不变的情况下避免重复计算，提高组件的性能。

`useRef：`用于在函数组件中保存可变值的引用。可以用于保存DOM元素的引用、保存上一次渲染的值等。

35、讲讲React的useState和useRef？

读懂 useState

`useState` 挂钩用于管理功能组件内的状态。它允许我们创建可以更新的变量，并在其值发生变化时触发重新渲染。 `useState` 钩子返回一个包含两个元素的数组：当前状态值和更新它的函数。

让我们使用 `useState` 创建一个简单的计数器组件：

```
js复制代码
import React, { useState } from 'react';

function Counter() {
  const [count, setCount] = useState(0);

  const increment = () => {
    setCount(count + 1);
  };

  return (
    <div>      <p>Count: {count}</p>      <button onClick={in
crement}>Increment</button>    </div>  );
}
```

在上面的代码中，我们使用数组解构语法来分配 `count` 状态变量和 `setCount` 函数来更新它。使用 `useState(0)` 将 `count` 的初始值设置为 0 。单击按钮时，将调用 `increment` 函数，通过添加 1 更新 `count` 状态。结果，组件重新渲染，反映 `count` 的更新值。

读懂 useRef

React 中的 `useRef` 钩子创建了一个在组件呈现之间持续存在的可变引用。与管理状态并触发重新渲染的 `useState` 不同， `useRef` 主要用于访问和操作 DOM 或存储不触发重新渲染的可变值。它返回一个带有 `current` 属性的可变对象。

假设我们想在单击按钮时关注输入字段。我们可以使用 `useRef` 来实现这一点，如下所示：

```
js复制代码
import React, { useRef } from 'react';

function MyComponent() {
  const inputRef = useRef(null);

  const handleClick = () => {
    inputRef.current.focus();
  };

  return (
    <div>      <input ref={inputRef} />      <button onClick=
{handleClick}>Focus Input</button>    </div>  );
}
```

在上面的示例中，我们使用 `useRef` 创建一个 `ref` 并将其分配给 `inputRef` 变量。我们将 `inputRef` 传递给输入元素的 `ref` 属性，使其可用于访问输入的 `DOM` 节点。单击按钮时，将执行 `handleClick` 函数，并调用 `inputRef.current.focus()` 以聚焦于输入字段。

比较

虽然 `useRef` 和 `useState` 都可以存储值，但它们有不同的用途：

`管理状态：` useState 旨在管理组件内的状态。当状态更新时，它会触发重新渲染，确保 UI 反映最新值。

`访问和操作 DOM：` useRef 主要用于与 DOM 交互，例如访问输入值或关注元素。它允许我们存储对 DOM 节点的引用并检索它们的属性，而无需触发重新渲染。

`跨渲染保留值：` useRef 在组件渲染之间维护相同的值，而 useState 在每次渲染期间初始化状态。

`重新渲染行为：`更新 useState 返回的值会导致组件重新渲染，同时更新使用 useRef 的 current 属性 不会触发重新渲染。

实际用例

`useRef` 用例：

- 访问 DOM 元素：当您需要访问或操作 DOM 元素（例如聚焦输入、滚动到特定元素或测量元素的大小）时， `useRef` 是合适的选择。它允许您创建对 DOM 节点的引用并访问其属性或方法。
- 存储可变值：如果您有一个值需要在渲染过程中保留，但不会影响组件的 UI 或触发重新渲染， `useRef` 是一个不错的选择。例如，您可以使用 `useRef` 存储以前的值、缓存值或保留可变值以进行比较。

`useState` 用例：

- 管理组件状态：当您需要管理和更新组件内的状态时，建议使用 `useState` 方法。它提供了一种存储和更新影响组件 UI 并触发重新渲染的值的方法。
- 处理用户交互：如果组件中有交互元素（例如复选框、输入字段或切换开关），则通常使用 `useState` 来管理与这些交互相关的状态。您可以根据用户输入更新状态并反映 UI 中的更改。

总结

总之， `useRef` 和 `useState` 都是React中必不可少的钩子，但它们有不同的用途。

`useRef` 主要用于访问和操作 DOM 或存储可变值而不触发重新渲染。它提供了一个在组件呈现之间持续存在的可变引用

`useState` 用于管理组件状态，当状态更新时触发重新渲染。它返回一个状态值和一个更新它的函数。

了解 `useRef` 和 `useState` 之间的差异并了解何时使用每个钩子对于编写有效且优化的 React 组件至关重要。通过正确利用 useRef 和 useState ，可以使用 React 构建交互式和高性能应用程序。

36、React父组件props变化时，子组件怎么监听？

1、通过 componentDidUpdate 钩子：

如果子组件是一个 class 组件，可以在 componentDidUpdate 钩子中比较新的 props 和之前的 props，然后做出相应的处理。当父组件的 props 更新时，React 会重新渲染子组件，触发 componentDidUpdate 钩子。

```
js复制代码
class ChildComponent extends React.Component {
  componentDidUpdate(prevProps) {
    if (this.props.someProp !== prevProps.someProp) {
      // Props 发生变化时执行某些操作    }
  }

  render() {
    return <div>{this.props.someProp}</div>;
  }
}
```

2、通过 useEffect 钩子：

如果子组件是一个函数组件，可以使用 `useEffect` 钩子监听父组件 props 的变化。通过在 `useEffect` 的依赖项中传入父组件的 props，可以在 props 变化时执行相应的逻辑。

```
js复制代码
import React, { useEffect } from 'react';

const ChildComponent = ({ someProp }) => {
  useEffect(() => {
    // 当 someProp 变化时执行某些操作  }, [someProp]);

  return <div>{someProp}</div>;
};
```

3、直接在 render 方法中比较：

在子组件的 `render` 方法中直接比较新的 props 和旧的 props，然后根据需要执行相应的操作。

```
js复制代码
class ChildComponent extends React.Component {
  render() {
    const { someProp } = this.props;
    const previousSomeProp = this.props.someProp;

    if (someProp !== previousSomeProp) {
      // Props 发生变化时执行某些操作    }

    return <div>{someProp}</div>;
  }
}
```

37、useMemo在React中怎么使用？

`useMemo` 是 React 中的一个 Hook，用于在渲染过程中执行**记忆化**计算，以避免在每次渲染时都重新计算。它接收一个函数和一个依赖数组，并返回函数的计算结果。在依赖项发生变化时，它会重新计算值。

项目中可以使用 `useMemo` 来优化性能，特别是当有昂贵的计算或引用相等性检查时。下面是 `useMemo` 的基本用法：

```
js复制代码
import React, { useMemo } from 'react';

const MyComponent = ({ a, b }) => {
  const result = useMemo(() => {
    // 执行一些昂贵的计算，或者进行一些引用相等性检查    return a +
b;
  }, [a, b]); // 依赖项数组，只有在 a 或者 b 变化时才会重新计算 resu
```

```
lt  return <div>{result}</div>;
};
```

在这个例子中，`useMemo` 接收一个函数作为第一个参数，该函数会在渲染过程中执行。在这个函数中，你可以进行一些昂贵的计算或引用相等性检查，并返回一个值。第二个参数是一个依赖项数组，只有当数组中的依赖项发生变化时，`useMemo` 才会重新计算值。在这个例子中，只有当 `a` 或 `b` 发生变化时，`result` 才会被重新计算。

需要注意的是，`useMemo` 只会在需要时才进行计算，因此它可以帮助优化性能，避免不必要的计算。但是，过度使用 `useMemo` 可能会导致代码变得复杂，因此应该根据实际情况进行权衡和使用。

38、React中React.Component 和 React.PureComponent的区别？

React.Component

`React.Component`是React中最常用的组件基类。它提供了组件生命周期的方法，如 `constructor`、`render`、`componentDidMount`、`shouldComponentUpdate`等。默认情况下，每次状态或`props`发生变化时，`React.Component`都会进行重新渲染。

```
js复制代码
class MyComponent extends React.Component {
  constructor(props) {
    super(props);
    this.state = { count: 0 };
  }

  increment = () => {
    this.setState(state => ({ count: state.count + 1 }));
  };

  render() {
    return (
      <div>        <p>Count: {this.state.count}</p>        <b
utton onClick={this.increment}>Increment</button>      </div>
```

```
);
  }
}
```

React.PureComponent

`React.PureComponent`是`React.Component`的一个子类，它在渲染行为上有所不同。`React.PureComponent`在默认情况下实现了`shouldComponentUpdate`方法，它通过浅比较组件的`props`和`state`来决定是否需要重新渲染。如果`props`和`state`都是相同的，组件将不会重新渲染。

```
js复制代码
class MyPureComponent extends React.PureComponent {
  // 无需手动实现shouldComponentUpdate，因为React.PureComponent已经提供了一个基于props和state的浅比较的实现  render() {
    // 渲染逻辑与MyComponent相同    return (
      <div>        <p>Count: {this.state.count}</p>        <button onClick={this.props.increment}>Increment</button>
</div>    );
  }
}
```

在这个例子中，MyPureComponent与MyComponent的渲染逻辑相同，但它继承自React.PureComponent，因此具有优化的渲染行为。

区别

- **性能优化：** React.PureComponent通过浅比较props和state来避免不必要的渲染，这可以提高组件的性能。如果组件的状态和属性在渲染之间没有变化，那么组件将不会重新渲染。
- **实现细节：** React.PureComponent在内部实现了shouldComponentUpdate方法，而React.Component需要开发者自己实现或继承默认行为（总是返回true）。
- **使用场景：** 当你确定组件的渲染结果只依赖于其props和state，并且这些props和state是可比较的（例如，基本类型或具有稳定引用的对象），那么使用React.PureComponent是一个好的选择。

39、Hooks相对Class的优化？

- **简洁性：** 使用 Hooks 可以使组件逻辑更加清晰和简洁。你可以将相关的逻辑拆分成多个独立的函数，而不必担心 Class 组件中的生命周期方法和状态管理会使代码变得复杂。
- **复用性：** Hooks 可以促进逻辑的重用。你可以编写自定义的 Hook 来提取组件中共享的状态逻辑，然后在不同的组件中重用这些 Hook。这种方式比较灵活，可以更好地组织和管理代码。
- **性能优化：** Hooks 可以帮助你更好地优化组件的性能。通过使用 useMemo、useCallback 等 Hook，可以避免不必要的渲染和函数重新创建，从而提高组件的性能。
- **逻辑复杂度：** 对于复杂的逻辑，使用 Hooks 可以更容易地管理组件的状态和副作用。相比于 Class 组件中需要处理多个生命周期方法和可能导致状态分散的情况，Hooks 提供了更统一的方式来管理组件的状态和副作用。
- **减少样板代码：** 使用 Hooks 可以减少组件中的样板代码。相比于 Class 组件中需要定义构造函数、绑定方法等，Hooks 提供了一种更简洁的方式来处理组件的逻辑。

40、setState 和 useState 的区别？

Class 组件中的 setState：

- `setState` 是 Class 组件中用于更新状态的方法。
- 在 Class 组件中，你需要手动管理状态的更新，调用 `setState` 来更新状态，并且可以传入一个函数作为参数，用于基于之前的状态计算新的状态。
- `setState` 可以接收一个对象或者函数作为参数，用于更新组件的状态，并且可以传入一个回调函数，在状态更新完成后执行一些操作。

函数组件中的 useState：

- `useState` 是 React Hooks 中的一个 Hook，用于在函数组件中添加状态。
- 在函数组件中使用 `useState` 声明状态变量，并返回一个状态变量和一个更新状态的函数。React 会在重新渲染时保留这个状态变量，并且更新它的值不会影响到其他状态变量。

- `useState` 只能接收一个初始状态作为参数，并且每次重新渲染时都会返回当前状态的最新值和更新状态的函数。

区别

- `setState` 是 Class 组件中用于更新状态的方法，而 `useState` 是函数组件中用于添加状态的 Hook。
- `setState` 可以传入对象或函数，并且可以在状态更新后执行回调函数，而 `useState` 只能接收初始状态作为参数，并且每次返回一个更新状态的函数。
- 使用 `useState` 可以使函数组件更加简洁和易于理解，而且更符合函数式编程的思想。

41、为什么不直接更新state？

在 React 中，状态(state)是组件的一个重要概念，用于管理组件的数据和状态变化。React 强烈建议不要直接修改 state，而是通过使用 setState 方法来进行状态的更新。下面我将解释为什么不要直接修改 state，并介绍如何正确地修改 state。

为什么不要直接修改 state？
直接修改 state 是不推荐的，主要有以下几个原因：

- **影响组件的更新机制：** React 使用虚拟 DOM 来管理组件的渲染和更新，通过比较新旧虚拟 DOM 的差异来最小化 DOM 操作，提高性能。如果直接修改 state，React 将无法感知到状态的变化，无法正确更新组件的渲染，可能导致 UI 不一致或性能下降。
- **可能导致不可预测的结果：** 直接修改 state 可能会破坏 React 的内部机制和约定，导致一些不可预测的问题。React 依赖于合成事件 (SyntheticEvent) 和批量更新机制来优化性能，直接修改 state 可能会绕过这些机制，引发问题。
- **破坏单向数据流：** React 推崇单向数据流的设计模式，通过将数据从父组件传递给子组件，保持数据的一致性和可追踪性。直接修改 state 可能导致数据流的混乱，增加代码的复杂性。

42、Hooks会取代render、props和高阶组件吗？

Hooks 并不会完全取代 `render`、`props` 和高阶组件，而是提供了一种更灵活的选择。

1. `render` **方法：**

- Hooks 并不取代 `render` 方法，而是在函数组件中直接返回 JSX，避免了使用 `render` 方法的需求。这使得组件的结构更加清晰，代码更加简洁。

2. **Props**：
   - 使用 Hooks 并不影响 props 的使用。无论是函数组件还是类组件，都可以通过 props 来传递数据和配置组件的行为。
3. **高阶组件**：
   - Hooks 并不完全取代高阶组件，但可以提供类似的功能。你可以使用自定义的 Hook 来提取和复用组件逻辑，从而达到类似高阶组件的效果，但更加简洁和灵活。

Hooks 的出现主要是为了解决 Class 组件中的一些问题，并提供一种更现代化、更灵活的方式来编写 React 组件。它们不是为了彻底取代已有的概念，而是为了补充和完善 React 的功能。因此，在使用 React 开发时，你可以根据具体情况选择合适的方式来编写组件，可以是 Class 组件、函数组件、Hooks 或者高阶组件，以满足项目的需求和开发者的偏好。

Hooks并不会完全取代 `render`、`props` 和高阶组件，而是提供了一种不同的方式来处理组件逻辑和状态。它们可以与现有的概念和模式共存，并根据具体情况选择最适合的方法来编写组件。Hooks提供了更灵活、更简洁的方式来管理组件状态、副作用和重用逻辑，但并不意味着完全取代了传统的 `render`、`props` 和高阶组件。

43、什么是纯函数？

纯函数是指具有以下两个特性的函数：

1. **确定性（Deterministic）**：对于相同的输入，纯函数始终会返回相同的输出，不会因为外部状态的变化而产生不同的结果。
2. **无副作用（No Side Effects）**：纯函数在执行过程中不会对除函数返回值以外的任何外部状态进行修改，包括不会修改全局变量、不会修改输入参数，也不会产生除函数返回值以外的其他可观察的影响。只专注于处理输入并返回输出。
   例如：

```
js复制代码
// 纯函数：计算两个数的和function add(a, b) {
  return a + b;
```

```
}

// 调用纯函数const result = add(3, 5);
console.log(result); // 输出 8
```

在这个例子中，`add` 函数接受两个参数 `a` 和 `b`，并返回它们的和。这个函数没有任何副作用，不会修改任何外部状态，只是简单地将两个数相加并返回结果。因此，它是一个纯函数。

由于纯函数的这些特性，它们在函数式编程中被广泛应用，并且具有一些重要的优点，如易于测试、并行化、缓存等。在编写代码时，尽可能地编写纯函数可以提高代码的可维护性和可预测性。

44、React为什么是单项数据流？有什么特征？

React采用单向数据流的设计模式，而不是双向数据绑定，主要有以下几个原因：

1. **易于理解和调试：** 在React应用程序中，数据流动方向是单向的，从父组件向子组件流动，这种设计模式使得应用程序的结构更加清晰、易于理解和调试。相比之下，双向数据绑定会导致数据流动方向不确定，增加了代码的复杂度和难度。
2. **更好的性能表现：** 在双向数据绑定中，当数据发生变化时，系统需要同时更新视图和数据模型，这会导致性能瓶颈。而在单向数据流中，数据的更新只会从父组件向子组件进行，这样可以避免不必要的视图更新，提高了应用程序的性能表现。
3. **更好的逻辑控制：** 在双向数据绑定中，由于数据的修改可能来自于多个组件，造成了数据的不可预测性。而在单向数据流中，数据的修改只能由父组件或本身进行，这样可以更好地控制应用程序的逻辑，减少了错误发生的概率。
4. **更容易实现服务端渲染：** React支持服务端渲染，这对于SEO和性能都非常重要。在双向数据绑定中，由于数据的修改可能来自于客户端，这使得服务端渲染变得更加困难。而在单向数据流中，由于数据的修改只能来自于服务端或本身，这样可以更方便地实现服务端渲染。

特征：

React 采用单向数据流的设计模式是为了简化数据流管理，降低系统复杂度，并提高组件的可维护性。单向数据流的特征包括：

1. **数据流向单一方向**：

- 数据从父组件流向子组件，形成了明确的数据流动路径。
- 子组件不可以直接修改父组件的数据，而是通过回调函数向父组件传递数据变化。

2. **易于追踪数据变化**：
   - 由于数据流向单一方向，当应用变得复杂时，易于跟踪数据的流向和变化。
   - 有利于调试和定位问题，因为数据的变化路径是可预测的。
3. **组件解耦**：
   - 组件之间通过 props 进行通信，而不是直接引用和修改彼此的状态。
   - 这种解耦可以使组件更加独立，便于重用和维护。
4. **可预测性**：
   - 单向数据流使得组件的状态变化更加可预测，因为数据只能通过特定的方式改变。
5. **性能优化**：
   - React 可以通过对数据流的追踪和优化来提高性能，因为数据的流向是可预测的，可以进行有效的优化措施。

45、类组件有状态，为什么还要用redux？

在React中，虽然类组件可以通过内部状态（`this.state`）来管理和维护状态，使用Redux的主要原因是它提供了一种集中管理应用全局状态的方式，尤其在大型应用中显得格外有用。以下是使用Redux相对于仅使用React类组件状态的一些优势：

集中的状态管理：

Redux提供一个全局的存储（store），用于集中管理应用的所有状态。这有助于不同组件间的状态共享，避免了将状态通过多层组件传递的复杂性。

可预测性和一致性：

Redux强制使用纯函数来执行所有状态更新（通过reducers），这意味着状态的改变是可预测的和可追踪的。这种方式有助于保证应用状态的一致性。

调试和开发工具：

Redux支持强大的开发工具（如Redux DevTools），允许开发者跟踪每个状态变化、进行时间旅行调试、以及状态回滚，这使得调试变得更加容易。

维护性和组织性：

在大型应用中，管理分散的状态可能变得复杂和困难。Redux通过规范化的数据流（单向数据流）和严格的代码结构，有助于提高代码的可维护性和组织性。

中间件和扩展功能：

Redux可以利用中间件来处理日志记录、异步API请求、复杂的同步逻辑等。这些中间件可以帮助管理副作用，并且提供比React组件状态更为强大的处理能力。

组件复用：

当状态管理从组件中抽离出来，组件本身变得更加轻量和专注于UI呈现，这使得组件更易于复用。

大型应用和团队开发：

在多人开发的大型项目中，Redux的结构化和规范化的模式可以帮助团队成员快速理解代码基础结构和业务逻辑，从而有助于团队协作。

46、redux遵循的三个原则是什么？

单一数据源

应用程序的整个状态应集中存储在一个Store中。这确保了数据的完整性和一致性，避免了因分散管理状态而产生的混乱。

状态是只读的

Store中的状态是不可变的，只能通过派发Action来修改。这保证了状态的完整性，防止意外修改。

纯函数来执行修改

为了描述 action 如何改变 state tree ，需要编写 reducers。reducer 只是一些纯函数，接收先前的 state 和 action，并返回新的 state。请注意，应该在 reducer 内部编写不可变的更新逻辑，而不是修改传入参数的值。

47、列出redux组件？

在redux中，核心组件包括`Action`、`Reducer`、`Store`和`Middleware`。

Action是一个普通的JavaScript对象，用于描述发生了什么事件。它必须包含一个type属性，用于标识事件的类型。可以在Action中添加其他自定义的属性来传递数据。

Reducer是一个纯函数，用于根据收到的Action来更新应用的状态（State）。它接收两个参数，当前的状态和收到的Action，然后返回一个新的状态。Reducer必须是纯函数，意味着给定相同的输入，它会始终返回相同的输出，而且不会有任何副作用。

Store是用于存储应用状态的地方。它是Redux应用的唯一数据源，可以通过getState方法获取当前状态，通过dispatch方法触发Action，通过subscribe方法注册监听器来监听状态的变化。

Middleware允许在dispatch一个Action到达reducer之前，对Action进行一些处理。它可以用来处理异步操作、日志记录、错误处理等。Middleware是通过包装store的dispatch方法来实现的。

`Reducer`的作用是根据收到的`Action`来更新应用的状态。它根据当前的状态和收到的`Action`返回一个新的状态。`Reducer`将多个小的`reducer`函数组合成一个根`reducer`函数，来管理整个应用的状态。每个`reducer`函数负责管理应用状态的一部分，然后根`reducer`将这些部分的状态合并成一个完整的应用状态。

48、数据如何通过redux流动？

Action 触发：

应用中的某个组件（如UI组件、网络请求、定时器等）触发一个 action。Action 是一个普通的 JavaScript 对象，必须包含一个 `type` 字段来描述动作类型，可以携带额外的数据作为 payload。

Action 到达 Reducer：

Action 被传递到 Reducer。Reducer 是一个纯函数，它接收之前的状态和当前的 action，并根据 action 的类型来生成新的状态。

State 更新：

Reducer 生成的新状态被存储在 Store 中，替换之前的旧状态。这个新的状态会被整个应用共享，成为新的“单一数据源”。

Store 通知订阅者：

Store 发送通知给所有订阅了它的组件或监听器，告知状态发生了变化。

组件重新渲染：

订阅了 Store 的组件会收到状态变化的通知，从而重新渲染并展示最新的状态。这保证了应用的视图与数据保持同步。

这个数据流程是单向的：`Action` 触发状态变化，`Reducer` 处理状态更新，`Store` 存储和通知，组件重新渲染。这种单向数据流的设计使得 `Redux` 应用具有可预测性、可维护性和可扩展性。

49、如何在rudux中定义Action？

在 Redux 中定义 Action 非常简单，一个 Action 实际上就是一个普通的 JavaScript 对象，它必须包含一个 `type` 字段来描述动作的类型。除了 `type` 字段之外，你可以在 Action 中包含其他的数据作为动作的载荷（payload），用于描述动作所需的具体信息。

通常，在 Redux 应用中，我们会定义一个函数来创建 Action 对象，这些函数被称为 Action Creator。Action Creator 是一个返回 Action 对象的函数，它可以接受参数并根据参数来创建不同的 Action 对象。

以下是在 Redux 中定义 Action 的步骤：

1、定义 Action 类型常量：

首先，定义一些 Action 类型常量，用来描述不同的动作类型。这些常量通常保存在单独的文件中，例如 `actionTypes.js`。

```
js复制代码
// actionTypes.jsexport const ADD_TODO = 'ADD_TODO';
export const DELETE_TODO = 'DELETE_TODO';
export const UPDATE_TODO = 'UPDATE_TODO';
```

2、编写 Action Creator 函数：

创建一个函数来生成 Action 对象，这个函数就是 Action Creator。

```
js复制代码
// actions.jsimport { ADD_TODO, DELETE_TODO, UPDATE_TODO } fr
om './actionTypes';

// Action Creator for adding a todoexport const addTodo = (te
xt) => ({
  type: ADD_TODO,
  payload: { text }
});

// Action Creator for deleting a todoexport const deleteTodo
= (id) => ({
  type: DELETE_TODO,
  payload: { id }
});

// Action Creator for updating a todoexport const updateTodo
= (id, newText) => ({
  type: UPDATE_TODO,
  payload: { id, newText }
});
```

3、使用 Action Creator：

在应用的其他地方（如组件、异步操作、定时器等）中调用这些 Action Creator 函数来创建具体的 Action 对象。

```
js复制代码
import { addTodo, deleteTodo, updateTodo } from './actions';

// Dispatching actions using Action Creatorsdispatch(addTodo
('Learn Redux'));
```

```
dispatch(deleteTodo(1));
dispatch(updateTodo(2, 'Redux is awesome'));
```

50、解释Reducer的作用？

Reducer 在 Redux 中扮演着关键的角色，它负责根据传入的 Action 来更新应用的状态。Reducer 是一个纯函数，它接收两个参数：之前的状态（state）和当前的 Action，然后返回一个新的状态。Reducer 的作用可以总结如下：

处理状态变化：

Reducer 接收到一个 Action 对象以及当前的状态，根据 Action 的类型和携带的数据（payload），对状态进行相应的处理和更新。

生成新的状态：

Reducer 应当返回一个全新的状态对象，而不是修改原始的状态对象。这保证了 Redux 中的状态是不可变的（Immutable），使得状态的变化可被追踪和记录。

保持纯函数特性：

Reducer 必须是纯函数，即给定相同的输入，始终产生相同的输出，并且没有副作用。这意味着 Reducer 在执行过程中不能修改参数，也不能执行与外部环境有关的操作，如发起网络请求或读取文件。

单一状态树：

Redux 应用中的状态被组织成一个单一的 JavaScript 对象，被称为“状态树”（state tree）。Reducer 负责管理这个状态树的不同部分，根据不同的 Action 类型来更新相应的状态片段。

可组合性和可扩展性：

Redux 应用通常包含多个 Reducer，每个 Reducer 负责管理状态树的一个部分。这些 Reducer 可以通过组合来创建一个根 Reducer，从而构建出整个应用的状态管理逻辑。这种设计使得 Redux 应用具有良好的可组合性和可扩展性。

Reducer 是 Redux 中负责处理状态变化的函数，它以一种可预测和可控的方式管理着应用的状态，确保状态的变化符合应用的业务逻辑，并且保持了状态的不变性和可追溯性。

51、store在redux中的意义是什么？redux有哪些优点？

意义

状态集中管理：

Redux 中的状态被组织成一个单一的状态树，存储在 Store 中。这种集中管理的方式使得状态的变化和访问变得更加直观和可控，方便了状态的跟踪和调试。

提供统一的状态管理接口：

通过 Store 提供的 `getState()` 方法，可以获取当前的应用状态；通过 `dispatch(action)` 方法，可以派发一个 Action 来触发状态的更新；通过 `subscribe(listener)` 方法，可以订阅状态的变化，以便在状态发生变化时执行相应的逻辑。

提供中间件支持：

Redux 提供了中间件机制，可以通过中间件来扩展 Redux 的功能，例如异步操作、日志记录、错误处理等。中间件可以拦截派发的 Action，对其进行处理，并最终将处理结果传递给下一个中间件或 Reducer。

保证状态不可变性：

Redux 要求状态是不可变的（Immutable），即状态在发生变化时不能直接修改原始对象，而是通过创建一个全新的状态对象来实现。Store 在这个过程中起到了关键作用，负责存储和更新不可变的状态对象。

提高代码可维护性：

Redux 的单向数据流和纯函数的设计理念使得应用的状态变化变得可预测和易于维护。通过明确的数据流动路径和统一的状态管理方式，减少了状态管理方面的混乱和错误，提高了代码的可读性和可维护性。

优点

可预测性：

Redux 的单向数据流和纯函数的设计保证了状态变化的可预测性，易于调试和测试。

可维护性：

通过集中式的状态管理和明确的数据流动路径，减少了代码中的混乱和错误，提高了代码的可读性和可维护性。

可扩展性：

Redux 的架构设计使得应用具有良好的可扩展性，可以通过组合 Reducer、使用中间件等方式来扩展应用的功能。

社区支持和生态系统：

Redux 是一个非常成熟和流行的状态管理库，拥有庞大的社区和丰富的生态系统，提供了大量的插件和工具来简化开发和优化性能。

与 React 结合紧密：

Redux 与 React 结合非常紧密，提供了专门的 React 绑定库（如 react-redux），可以方便地在 React 应用中使用 Redux 进行状态管理。

52、React Router与常规路由有何不同？

前端路由：

React Router 是一种前端路由解决方案，它在客户端处理路由，而不是像传统的后端路由那样由服务器处理。这意味着页面之间的切换是在浏览器中进行的，不需要重新加载整个页面，从而提升了用户体验。

单页面应用（SPA）支持：

React Router 特别适用于单页面应用（SPA），它能够在不刷新页面的情况下，根据 URL 的变化来动态渲染不同的组件，实现页面间的无缝切换。

声明式路由配置：

React Router 提供了一种声明式的方式来配置路由，通过 JSX 元素来定义路由和组件之间的映射关系，使得路由配置更加直观和易于理解。

组件化：

React Router 将路由信息视为组件树的一部分，路由器本身也是一个 React 组件。这种组件化的设计使得路由与应用的其他部分更加紧密地集成在一起，提高了组件的复用性和可维护性。

动态路由匹配：

React Router 支持动态路由匹配，可以通过参数化路由来处理不同路径下的相似页面，从而更加灵活地处理不同的路由情况。

53、你是怎样理解“在React中，一切都是组件”？这句话怎样解释React中render()的目的？

在 React 中，一切都是组件”是指在 React 应用中，所有的 UI 都可以抽象为组件的形式。无论是页面的整体布局、还是页面中的特定元素，都可以用组件来表示和管理。这种组件化的设计思想使得 React 应用的开发更加模块化、灵活和可维护。

在 React 中，每个组件都是一个 JavaScript 类或函数，它们接受输入的属性（props），并返回一个描述页面展示内容的 React 元素树。这些组件可以相互组合、嵌套，形成一个复杂的应用界面。

`render()` 方法是 React 组件类中的一个生命周期方法，它的作用是根据组件的状态（state）和属性（props）来生成 Virtual DOM，并最终将 Virtual DOM 渲染到页面上。`render()` 方法返回的是一个描述组件 UI 的 React 元素，它描述了组件在特定状态下应该呈现的样子。

因此，`render()` 方法的目的是将组件的 UI 描述转换为真实的页面元素，从而实现组件在页面上的渲染。通过调用 `render()` 方法，React 将根据组件的状态和属性动态地生成 UI，并确保 UI 的实时更新和同步渲染。

54、React中的箭头函数是什么？怎么用？

在 React 中，箭头函数是一种 JavaScript 的函数定义方式，它提供了一种简洁的语法来声明函数，并且在函数内部自动绑定了 `this` 关键字。

箭头函数的语法形式如下：

```
js复制代码
const myFunction = (arg1, arg2) => {
  // 函数体};
```

箭头函数可以有零个或多个参数，当只有一个参数时，可以省略参数周围的括号。如果函数体只有一条语句，可以省略大括号和 `return` 关键字，并且该语句的结果将作为函数的返回值。例如：

```
js复制代码
const add = (a, b) => a + b; // 省略了大括号和 return
```

箭头函数与传统的函数定义方式相比，具有以下几点特性：

- **简洁性：** 箭头函数的语法更加简洁明了，特别是在只有单条语句的情况下，可以更加精简地表达函数逻辑。
- **词法绑定的 `this`：** 箭头函数的 `this` 值在定义时确定，而不是在调用时确定，它会捕获其所在上下文的 `this` 值，因此不会受到函数调用方式的影响，通常用来避免传统函数中 `this` 绑定问题的困扰。

在 React 中，箭头函数经常用于定义组件的方法，例如事件处理函数、生命周期方法等，以确保 `this` 指向组件实例，而不受调用方式的影响。

55、redux与mobx的区别？

Redux 和 MobX 都是用于管理应用状态的 JavaScript 库，但它们在设计理念和使用方式上有一些不同之处。

1. **设计理念：**
   - Redux 的设计理念是基于函数式编程的思想，它提倡使用不可变的数据结构来描述应用的状态变化，并通过纯函数来处理状态的修改和更新。
   - MobX 则更加倾向于面向对象的编程范式，它允许你定义可观察的状态对象，并在需要时直接修改这些对象，然后 MobX 会自动追踪状态变化并触发相关的更新。
2. **数据流管理：**
   - 在 Redux 中，应用的状态被统一管理在一个单一的 Store 中，通过派发 Action 来触发状态的修改，然后通过纯函数的方式来处理 Action，最终更新 Store 中的状态。
   - 而在 MobX 中，状态是分散存储在各个可观察的对象中的，当对象的状态发生改变时，相关的观察者会被自动通知并执行相应的更新。
3. **使用方式：**

- Redux 更加显式和规范，需要定义 Action、Reducer、Store 等概念，并且通常需要使用中间件来处理异步操作。
- MobX 则更加简单直观，只需要定义可观察的状态对象，并在需要时直接修改状态即可，不需要定义额外的概念或中间件。

4. **性能：**
   - Redux 的状态更新是通过纯函数来处理的，每次状态更新都会产生新的状态对象，因此需要进行深比较来判断是否需要重新渲染组件，可能会影响性能。
   - MobX 则采用了更加智能的方式来追踪状态变化，只会重新渲染受影响的组件，因此在某些情况下可能会更加高效。