09-React全家桶实战-常见面试题

https://juejin.cn/post/6940942549305524238

React基础

1 什么是React?

- React是Facebook在2013年开源的JavaScript框架。
- 官方对它的解释是：用于构建用户界面的 JavaScript 库。
- 现今React和Vue是国内最为流行的两个框架，都是帮助我们来构建用户界面的JavaScript库。

React的特点和优势

React的特点：

- 声明式编程：
  - 声明式编程是目前整个大前端开发的模式：Vue、React、Flutter、SwiftUI；
  - 它允许我们只需要维护自己的状态，当状态改变时，React可以根据最新的状态去渲染我们的UI界面；
- 组件化开发：
  - 组件化开发页面目前前端的流行趋势，我们会讲复杂的界面拆分成一个个小的组件；
  - 如何合理的进行组件的划分和设计也是后面我会讲到的一个重点；
- 多平台适配：
  - 2013年，React发布之初主要是开发Web页面；
  - 2015年，Facebook推出了ReactNative，用于开发移动端跨平台；（虽然目前Flutter非常火爆，但是还是有很多公司在使用ReactNative）；
  - 2017年，Facebook推出ReactVR，用于开发虚拟现实Web应用程序；（随着5G的普及，VR也会是一个火爆的应用场景）；

React的优势

React由Facebook来更新和维护，它是大量优秀程序员的思想结晶：

- React的流行不仅仅局限于普通开发工程师对它的认可，大量流行的其他框架借鉴React的思想；

Vue.js框架设计之初，有很多的灵感来自Angular和React。

- 包括Vue3很多新的特性，也是借鉴和学习了React
- 比如React Hooks是开创性的新功能（也是我们课程的重点）
- Vue Function Based API学习了React Hooks的思想

Flutter的很多灵感都来自React，Flutter中的Widget – Element – RenderObject，对应的就是JSX – 虚拟DOM – 真实DOM。

所以，可以说React是前端的先驱者，它会引领整个前端的潮流。

4.什么是JSX?

- JSX是一种JavaScript的语法扩展（eXtension），也在很多地方称之为JavaScript XML，因为看起就是一段XML语法；
- 它用于描述我们的UI界面，并且其完全可以和JavaScript融合在一起使用；
- 它不同于Vue中的模块语法，你不需要专门学习模块语法中的一些指令（比如v-for、v-if、v-else、v-bind）；

JSX转换的本质是什么?

- 实际上，jsx 仅仅只是 `React.createElement(component, props, ...children)` 函数的语法糖。
- 所有的jsx最终都会被转换成 `React.createElement` 的函数调用。

为什么React选择了JSX?

- React认为渲染逻辑本质上与其他UI逻辑存在内在耦合
- - 比如UI需要绑定事件（button、a原生等等）；
  - 比如UI中需要展示数据状态，在某些状态发生改变时，又需要改变UI；
- 他们之间是密不可分，所以React没有将标记分离到不同的文件中，而是将它们组合到了一起，这个地方就是组件（Component）；

React为什么采用虚拟DOM?

为什么要采用虚拟DOM，而不是直接修改真实的DOM呢?

- 很难跟踪状态发生的改变：原有的开发模式，我们很难跟踪到状态发生的改变，不方便针对我们应用程序进行调试；
- 操作真实DOM性能较低：传统的开发模式会进行频繁的DOM操作，而这一的做法性能非常的低；
- 虚拟DOM帮助我们从命令式编程转到了声明式编程的模式。
- 虚拟DOM有利于实现跨平台的能力，即一套代码可以打包出各个平台的应用。

Raect事件函数绑定this有几种方式?

- **方案一：bind给btnClick显示绑定this**

  在传入函数时，我们可以主动绑定this：

  ```
  <button onClick={this.btnClick.bind(this)}>点我一下(React)</button>
  ```

- **方案二：使用 ES6 class fields 语法**

  你会发现我这里将btnClick的定义变成了一种赋值语句

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

    this.state = {
      message: "你好啊,李银河"
    }
  }

  render() {
    return (
      <div>
        <button onClick={this.btnClick}>点我一下(React)</button>
        <button onClick={this.btnClick}>也点我一下(React)</button>
      </div>
    )
  }

  btnClick = () => {
    console.log(this);
    console.log(this.state.message);
  }
}
```

方案三：事件监听时传入箭头函数（推荐）

因为 `onClick` 中要求我们传入一个函数，那么我们可以直接定义一个箭头函数传入

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

    this.state = {
      message: "你好啊,李银河"
    }
  }

  render() {
    return (
      <div>
        <button onClick={() => this.btnClick()}>点我一下(React)</button>
        <button onClick={() => this.btnClick()}>也点我一下(React)</button>
      </div>
    )
  }

  btnClick() {
    console.log(this);
    console.log(this.state.message);
```

```
  }
}
```

说说事件的参数如何传递?

在执行事件函数时，有可能我们需要获取一些参数信息：比如event对象、其他参数

情况一：获取event对象。

很多时候我们需要拿到event对象来做一些事情（比如阻止默认行为）

假如我们用不到this，那么直接传入函数就可以获取到event对象。

```
class App extends React.Component {
  constructor(props) {

  render() {
    return (
      <div>
        <a href="http://www.baidu.com" onClick={this.btnClick}>点我一下</a>
      </div>
    )
  }

  btnClick(e) {
    e.preventDefault();
    console.log(e);
  }
}
```

情况二：获取更多参数。 有更多参数时，我们最好的方式就是传入一个箭头函数，主动执行的事件函数，并且传入相关的其他参数；

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

    this.state = {
      names: ["衣服", "鞋子", "裤子"]
    }
  }

  render() {
    return (
      <div>
        <a href="http://www.baidu.com" onClick={this.aClick}>点我一下</a>

        {
```

```
          this.state.names.map((item, index) => {
            return (
              <a href="#" onClick={e => this.aClick(e, item, index)}>{item}
</a>
            )
          })
        }
      </div>
    )
  }

  aClick(e, item, index) {
    e.preventDefault();
    console.log(item, index);
    console.log(e);
  }
}
```

React的事件和普通的HTML事件有什么不同?

区别:

- 对于事件名称命名方式，原生事件为全小写，react 事件采用小驼峰;
- 对于事件函数处理语法，原生事件为字符串，react 事件为函数;
- react 事件不能采用 return false 的方式来阻止浏览器的默认行为，而必须要地明确地调用 `preventDefault()` 来阻止默认行为。

合成事件是 react 模拟原生 DOM 事件所有能力的一个事件对象，其优点如下:

- 兼容所有浏览器，更好的跨平台;
- 将事件统一存放在一个数组，避免频繁的新增与删除（垃圾回收）。
- 方便 react 统一管理和事务机制。

事件的执行顺序为原生事件先执行，合成事件后执行，合成事件会冒泡绑定到 document 上，所以尽量避免原生事件与合成事件混用，如果原生事件阻止冒泡，可能会导致合成事件不执行，因为需要冒泡到 document 上合成事件才会执行。

什么是高级函数？什么是高阶HOC组件?

什么是高阶组件呢？相信很多同学都听说过，也用过 `高阶函数`，它们非常相似，所以我们可以先来回顾一下什么是 `高阶函数`。

高阶函数的维基百科定义：至少满足以下条件之一:

- 接受一个或多个函数作为输入;
- 输出一个函数;

JavaScript中比较常见的filter、map、reduce都是高阶函数。

那么说明是高阶组件呢?

- 高阶组件的英文是 **Higher-Order Components**，简称为 `HOC`；
- 官方的定义：**高阶组件是参数为组件，返回值为新组件的函数**；

我们可以进行如下的解析：

- 首先，`高阶组件` 本身不是一个组件，而是一个函数；
- 其次，这个函数的参数是一个组件，返回值也是一个组件；

高阶组件的调用过程类似于这样：

```
const EnhancedComponent = higherOrderComponent(WrappedComponent);
```

高阶函数的编写过程类似于这样：

```
function higherOrderComponent(WrapperComponent) {
  return class NewComponent extends PureComponent {
    render() {
      return <WrapperComponent/>
    }
  }
}
```

高阶组件并不是React API的一部分，它是基于React的组合特性而形成的设计模式；

高阶组件在一些React第三方库中非常常见：

- 比如redux中的connect；
- 比如react-router中的withRouter；

说说高阶组件的应用场景?

- props的增强，不修改原有代码的情况下，添加新的props。

假如我们有如下案例：

```
class Header extends PureComponent {
  render() {
    const { name, age } = this.props;
    return <h2>Header {name + age}</h2>
  }
}

export default class App extends PureComponent {
  render() {
    return (
      <div>
        <Header name="aaa" age={18} />
```

```
        </div>
      )
    }
  }
```

可以通过一个高阶组件，让使用者在不破坏原有结构的情况下对某个组件增强props

```
function enhanceProps(WrapperCpn, otherProps) {
 return props => <WrapperCpn {...props} {...otherProps} />
}

const EnhanceHeader = enhanceProps(Header, {height: 1.88})
```

- 利用高阶组件来共享Context

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

const UserContext = createContext({
  nickname: "默认",
  level: -1
})

function Header(props) {
  return (
    <UserContext.Consumer>
      {
        value => {
          const { nickname, level } = value;
          return <h2>Header {"昵称:" + nickname + "等级" + level}</h2>
        }
      }
    </UserContext.Consumer>
  )
}

function Footer(props) {
  return (
    <UserContext.Consumer>
      {
        value => {
          const { nickname, level } = value;
          return <h2>Footer {"昵称:" + nickname + "等级" + level}</h2>
        }
      }
    </UserContext.Consumer>
  )
}
```

```
const EnhanceHeader = enhanceProps(Header, { height: 1.88 })

export default class App extends PureComponent {
  render() {
    return (
      <div>
        <UserContext.Provider value={{ nickname: "why", level: 90 }}>
          <Header />
          <Footer />
        </UserContext.Provider>
      </div>
    )
  }
}
```

React如何创建refs来获取对应的DOM呢?

如何创建refs来获取对应的DOM呢？目前有三种方式：

方式一：传入字符串

- 使用时通过 `this.refs.传入的字符串` 格式获取对应的元素；

方式二：传入一个对象

- 对象是通过 `React.createRef()` 方式创建出来的；
- 使用时获取到创建的对象其中有一个 `current` 属性就是对应的元素；

方式三：传入一个函数

- 该函数会在DOM被挂载时进行回调，这个函数会传入一个 元素对象，我们可以自己保存；
- 使用时，直接拿到之前保存的元素对象即可；

对 React context 的理解

在React中，数据传递一般使用props传递数据，维持单向数据流，这样可以让组件之间的关系变得简单且可预测，但是单项数据流在某些场景中并不适用。单纯一对的父子组件传递并无问题，但要是组件之间层层依赖深入，props就需要层层传递显然，这样做太繁琐了。

Context 提供了一种在组件之间共享此类值的方式，而不必显式地通过组件树的逐层传递 props。

可以把context当做是特定一个组件树内共享的store，用来做数据传递。**简单说就是，当你不想在组件树中通过逐层传递props或者state的方式来传递数据时，可以使用Context来实现跨层级的组件数据传递。**

JS的代码块在执行期间，会创建一个相应的作用域链，这个作用域链记录着运行时JS代码块执行期间所能访问的活动对象，包括变量和函数，JS程序通过作用域链访问到代码块内部或者外部的变量和函数。

假如以JS的作用域链作为类比，React组件提供的Context对象其实就好比一个提供给子组件访问的作用域，而 Context对象的属性可以看成作用域上的活动对象。由于组件 的 Context 由其父节点链上所有组件通 过 getChildContext（）返回的Context对象组合而成，所以，组件通过Context是可以访问到其父组件链上所有节点组件提供的Context的属性。

类组件与函数组件有什么异同?

相同点： 组件是 React 可复用的最小代码片段，它们会返回要在页面中渲染的 React 元素。也正因为组件是 React 的最小编码单位，所以无论是函数组件还是类组件，在使用方式和最终呈现效果上都是完全一致的。

我们甚至可以将一个类组件改写成函数组件，或者把函数组件改写成一个类组件（虽然并不推荐这种重构行为）。从使用者的角度而言，很难从使用体验上区分两者，而且在现代浏览器中，闭包和类的性能只在极端场景下才会有明显的差别。所以，基本可认为两者作为组件是完全一致的。

不同点：

- 它们在开发时的心智模型上却存在巨大的差异。类组件是基于面向对象编程的，它主打的是继承、生命周期等核心概念；而函数组件内核是函数式编程，主打的是 immutable、没有副作用、引用透明等特点。
- 之前，在使用场景上，如果存在需要使用生命周期的组件，那么主推类组件；设计模式上，如果需要使用继承，那么主推类组件。但现在由于 React Hooks 的推出，生命周期概念的淡出，函数组件可以完全取代类组件。其次继承并不是组件最佳的设计模式，官方更推崇“组合优于继承”的设计概念，所以类组件在这方面的优势也在淡出。
- 性能优化上，类组件主要依靠 shouldComponentUpdate 阻断渲染来提升性能，而函数组件依靠 React.memo 缓存渲染结果来提升性能。
- 从上手程度而言，类组件更容易上手，从未来趋势上看，由于React Hooks 的推出，函数组件成了社区未来主推的方案。
- 类组件在未来时间切片与并发模式中，由于生命周期带来的复杂度，并不易于优化。而函数组件本身轻量简单，且在 Hooks 的基础上提供了比原先更细粒度的逻辑组织与复用，更能适应 React 的未来发展。

Redux

什么是Redux?

- Redux 是 JavaScript应用程序的可预测状态容器。
- 它可以帮助您编写行为一致、在不同环境（客户端、服务器和本机）中运行并且易于测试的应用程序。最重要的是，它提供了出色的开发人员体验，例如实时代码编辑与时间旅行调试器相结合。
- 可以将 Redux 与React或任何其他视图库一起使用。它很小（2kB，包括依赖项），但有一个庞大的插件生态系统可用。

为什么需要redux?

JavaScript开发的应用程序，已经变得越来越复杂了：

- JavaScript需要管理的状态越来越多，越来越复杂；

- 这些状态包括服务器返回的数据、缓存数据、用户操作产生的数据等等，也包括一些UI的状态，比如某些元素是否被选中，是否显示加载动效，当前分页；

管理不断变化的state是非常困难的：

- 状态之间相互会存在依赖，一个状态的变化会引起另一个状态的变化，View页面也有可能会引起状态的变化；
- 当应用程序复杂时，state在什么时候，因为什么原因而发生了变化，发生了怎么样的变化，会变得非常难以控制和追踪；

React是在视图层帮助我们解决了DOM的渲染过程，但是State依然是留给我们自己来管理：

- 无论是组件定义自己的state，还是组件之间的通信通过props进行传递；也包括通过Context进行数据之间的共享；
- React主要负责帮助我们管理视图，state如何维护最终还是我们自己来决定；

Redux就是一个帮助我们管理State的容器，提供了可预测的状态管理。

Redux除了和React一起使用之外，它也可以和其他界面库一起来使用（比如Vue），并且它非常小（包括依赖在内，只有2kb）

redux的三大原则是什么?

单一数据源

整个应用程序的state被存储在一颗object tree中，并且这个object tree只存储在一个 store 中：

- Redux并没有强制让我们不能创建多个Store，但是那样做并不利于数据的维护；
- 单一的数据源可以让整个应用程序的state变得方便维护、追踪、修改；

State是只读的

唯一修改State的方法一定是触发action，不要试图在其他地方通过任何的方式来修改State：

- 这样就确保了View或网络请求都不能直接修改state，它们只能通过action来描述自己想要如何修改state；
- 这样可以保证所有的修改都被集中化处理，并且按照严格的顺序来执行，所以不需要担心race condition（竟态）的问题；

使用纯函数来执行修改

通过reducer将 `旧state` 和 `actions` 联系在一起，并且返回一个新的State：

- 随着应用程序的复杂度增加，我们可以将reducer拆分成多个小的reducers，分别操作不同state tree的一部分；
- 但是所有的reducer都应该是纯函数，不能产生任何的副作用；

React Router

什么是React Router

- React Router是一个强大的路由库，建立在React的基础上，可以帮助向应用程序添加新的屏幕和

流程。
- 这样可以使URL与网页上显示的数据保持同步。
- 它保持标准化的结构和行为，并用于开发单页Web应用程序。React Router有一个简单的API。

页面传递参数有几种方式?

传递参数有三种方式：

- 动态路由的方式；
- search传递参数；
- to传入对象；

动态路由的方式

动态路由的概念指的是路由中的路径并不会固定：

- 比如`/detail`的path对应一个组件Detail；
- 如果我们将path在Route匹配时写成`/detail/:id`，那么`/detail/abc`、`/detail/123`都可以匹配到该Route，并且进行显示；
- 这个匹配规则，我们就称之为动态路由；

通常情况下，使用动态路由可以为路由传递参数。

```
<div>
 ...其他Link
  <NavLink to="/detail/abc123">详情</NavLink>

  <Switch>
    ... 其他Route
    <Route path="/detail/:id" component={Detail}/>
    <Route component={NoMatch} />
  </Switch>
</div>
```

detail.js的代码如下：

- 我们可以直接通过match对象中获取id；
- 这里我们没有使用withRouter，原因是因为Detail本身就是通过路由进行的跳转；

```
import React, { PureComponent } from 'react'

export default class Detail extends PureComponent {
  render() {
    console.log(this.props.match.params.id);

    return (
      <div>
        <h2>Detail: {this.props.match.params.id}</h2>
      </div>
    )
  }
}
```

search传递参数

NavLink写法：

- 我们在跳转的路径中添加了一些query参数；

```
<NavLink to="/detail2?name=why&age=18">详情2</NavLink>

<Switch>
  <Route path="/detail2" component={Detail2}/>
</Switch>
```

Detail2中如何获取呢?

- Detail2中是需要在location中获取search的；
- 注意：这个search没有被解析，需要我们自己来解析；

```
import React, { PureComponent } from 'react'

export default class Detail2 extends PureComponent {
  render() {
    console.log(this.props.location.search); // ?name=why&age=18

    return (
      <div>
        <h2>Detail2:</h2>
      </div>
    )
  }
}
```

to传入对象

to可以直接传入一个对象

```
<NavLink to={{
    pathname: "/detail2",
    query: {name: "kobe", age: 30},
    state: {height: 1.98, address: "洛杉矶"},
    search: "?apikey=123"
  }}>
  详情2
</NavLink>
```

获取参数:

```
import React, { PureComponent } from 'react'

export default class Detail2 extends PureComponent {
  render() {
    console.log(this.props.location);

    return (
      <div>
        <h2>Detail2:</h2>
      </div>
    )
  }
}
```