09-React全家桶实战-常见面试题

React基础

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。

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开发的三个依赖包是什么？分别有什么作用?

- react:包含react所有必须的核心代码
- react-dom：react 渲染在不同平台所需要的核心代码
- babel:将jsx转换为 React 代码的工具

React如何进行列表数据的展示?

- 方式一：直接使用for循环

```
    render() {
        // 1.对movies进行for循环
        const liEls = []
        for (let i = 0; i < this.state.movies.length; i++) {
        const movie = this.state.movies[i]
        const liEl = <li>{movie}</li>
        liEls.push(liEl)
        }
          <div>
```

```
            <h2>电影列表</h2>
            <ul>
              <li> {liEls}</li>
            </ul>
          </div>
        )
      }
```

- 方式二：map 高阶函数

```
      render()
#返回一个新的数组
        const liEls = this.state.movies.map(movie => <li>{movie}</li>)

        return (
          <div>
            <h2>电影列表</h2>
            <ul>
              <li> {liEls}</li>
            </ul>
          </div>
        )
      }
```

- 方式三： map 高阶函数 表达式

```
<ul>
{this.state.movies.map(movie => <li>{movie}</li>)}
</ul>
```

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组件可以如何进行划分?

- 根据定义方式
  - 函数组件
  - 类组件
- 根据组件内部有无状态需要维护
  - 无状态组件
  - 有状态组件
- 根据组件的不同职责
  - 展示型组件
  - 容器型组件
- 关注UI的展示
  - 函数组件,无状态组件,展示型组件
- 关注数据逻辑的展示
  - 类组件,有状态组件,容器型组件

React重要的组件生命周期有哪些?

- componentDidMount
  - 会在组件挂载后立即调用
  - 操作DOM
  - 发送网络请求
- componentDidUpdate
  - 会在组件更新后立即调用
  - 首次渲染不会执行此方法
  - 可以在组件更新后操作DOM
- conponentWillUnmount

    - 会在组件卸载及销毁前调用
    - 清除定时器
    - 清除事件监听

React中如何实现父子组件间的通信?

- 父传子
    - 在父组件通过属性传递数据给子组件
    - 子组件通过props拿到传递进来的内容
- 子传父
    - 通过传递函数, 调用回调函数（这种方式可以实现类似于Vue的插槽）
    - 通过函数的参数传递要传递的数据

React中非父子组件的通信有哪些方式?

- 事件总线EventBus
- 使用context上下文
    - 创建context
    - 在要使用的组件, 一般是根组件导入context
    - 使用<context.Provider>包裹后代组件
    - 在要使用的后代组件引入 context
        - xxxx.contextType = context
        - 在render方法中可以通过this.context拿到传递过来的值
- 使用Redux全局状态管理

React中有插槽的概念吗?

- React中是没有插槽的概念
- 实现插槽效果的方式
    - 在子组件中通过props.children拿到需要父组件嵌套的内容
    - 接着将拿到的内容插入到jsx中进行展示
- 通过props将要展示的内容, 通过函数的方式传递给子组件
    - 子组件通过props拿到传递进来的函数，接着调用该函数拿到要展示的内容，并将内存插入到jsx中展示

React为什么要强调不可变的力量？如何实现不可变的力量?

- 不可变的力量:
    - 不要直接去修改this.state中的值(主要指对象),
    - 若是想修改的话, 应该是将这整个值全部修改掉
    - 注意: 值类型, 在修改的时候, 本身就全部替换掉了, 所以不需要其他操作, 直接改就可以
- 实现:

- 将对象浅拷贝赋值给一个新的变量,
- 再将这个新的变量赋值给this.state中的值

React中获取DOM的方式有哪些?

- ref获取DOM

```
getDOM() {
// 方式一: 在react元素上绑定ref字符串 - 这种方式react已经不推荐了
// console.log(this.refs.http)

// 方式二: 提前创建好ref对象, createRef(), 将创建出来的对象绑定到元素(推荐)
// console.log(this.titleRef.current)

// 方式三: 传入一个回调函数, 在对应的元素被渲染之后, 回调函数被执行, 并且将元素传入
(16.3之前的版本)
// console.log(this.titleEl)
}
<h3 ref="http">大大怪将军</h3>
<h3 ref={this.titleRef}>小小怪下士</h3>
<h3 ref={el => this.titleEl = el}>瑞克</h3>
<button onClick={() => this.getDOM()}>获取DOM</button>
```

- ref获取组件实例 -- `createRef`

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

constructor() {
    super()
    this.state = {}
    this.HWRef = createRef()
}

getComponent() {
    console.log(this.HWRef.current)
    this.HWRef.current.test()
}

<HelloWorld ref={this.HWRef} />
<button onClick={() => this.getComponent()}>获取组件实例</button>
```

- ref获取函数组件 -- 函数式组件是没有实例的，所以无法通过ref获取他们的实例 -- `React.forwardRef`

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

```
const HelloWorld = forwardRef(function(props, ref) {
  return (
    <div>
      <h2 ref={ref}>函数组件</h2>
      <h4>大大怪将军</h4>
    </div>
  )
})
constructor() {
    super()
    this.state = {}
    this.HWRef = createRef()
}

getComponent() {
    console.log(this.HWRef.current)
}

render() {
    return (
      <div>
        <HelloWorld ref={this.HWRef} />
        <button onClick={() => this.getComponent()}>获取DOM</button>
      </div>
    )
}
```

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。

什么是受控组件和非受控组件?

- 受控组件
  - 在 React 中，可变状态通常保存在组件的 state 属性中，并且只能通过使用 setState()来更新
  - 我们将两者结合起来，使React的state成为“唯一数据源”
  - 渲染表单的 React 组件还控制着用户输入过程中表单发生的操作
  - 被 React 以这种方式控制取值的表单输入元素就叫做“受控组件”

```
this.state = {
  message: ""
}
changeInput(event) {
  console.log(event.target.value)
  this.setState({ message: event.target.value })
}

render(){
   <input type="text" value={message} onChange={(event) =>
this.changeInput(event)} />
}
```

- 非受控组件
  - 在受控组件中，表单数据是由 React 组件来管理的
  - 非受控组件中，表单数据将交由 DOM 节点来处理

```
this.messageRef.current.value

// 在非受控组件中通常使用defaultValue来设置默认值
render(){
    <input type="text" defaultValue={message} ref={this.messageRef} />
}
```

什么是Fragment，有什么作用?

- Fragment 允许将子列表分组，而无需向 DOM 添加额外节点；
- 它的简写看起来像空标签 <> </>
- 如果我们需要在Fragment中添加key，那么就不能使用短语法
- Fragment类似于Vue中的template，类似与小程序中的block

React进阶

什么是虚拟DOM？虚拟DOM在React中起到什么作用?

- 什么是虚拟DOM?
  - Virtual DOM 是一种编程概念，UI以一种理想化或者说虚拟化的方式保存在内存中
  - Virtual DOM 本质上是 JavaScript 对象，是真实 DOM 的描述，用一个 JS 对象来描述一个 DOM 节点
  - 我们知道jsx转成React代码的本质是 - 转换成React.createElement的函数调用
  - 通过React.createElement的函数创建出来的`ReactElement`对象
  - React利用`ReactElement`对象组成了一个JavaScript的对象树 - JavaScript的对象树就是**虚拟DOM**
- 虚拟DOM在React中的作用
  - 虚拟DOM 通过diff算法 - 以最小的代价更新变化的视图
  - 跨平台渲染
  - 声明式编程 - 虚拟DOM帮助我们从命令式编程转到了声明式编程的模式
    - 你只需要告诉React希望让UI是什么状态
    - React来确保DOM和这些状态是匹配的
    - 不需要直接进行DOM操作，就可以从手动更改DOM、属性操作、事件处理中解放出来

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。

React的diff算法和key的作用

- React的渲染流程
  - 在render函数中返回jsx, jsx会创建出`ReactElement`对象(通过React.createElement的函数创建出来的)
  - `ReactElement`最终会形成一颗树结构, 这颗树结构就是vDOM
  - React会根据这样的vDOM渲染出真实DOM
- React更新流程
  - props/state发生改变
  - render函数重新执行
  - 产生新的DOM树结构
  - 新旧DOM树 进行diff算法
  - 计算出差异进行更新
  - 最后更新到真实DOM

什么是diff算法?
diff算法并非React独家首创,但是React针对diff算法做了自己的优化,使得时间复杂度优化成了O(n)
对比两颗树结构,然后帮助我们计算出vDOM中真正变化的部分,并只针对该部分进行实际的DOM操作,而非渲染整个页面,从而保证了每次操作后页面的高效渲染。

- React在props或state发生改变时，会调用React的render方法，会创建一颗不同的树
- React需要基于这两颗不同的树之间的差别来判断如何有效的更新UI
- 如果一棵树参考另外一棵树进行完全比较更新，那么即使是最先进的算法，该算法的**复杂程度为O(n³)**，其中 n 是树中元素的数量
- 如果在 React 中使用了该算法，那么展示 1000 个元素所需要执行的计算量将在十亿的量级范围
- 这个开销太过昂贵了，于是，React对这个算法进行了优化，**将其优化成了O(n)**
  - 同层节点之间相互比较，不会垮节点比较(一旦某个节点不同,那么包括其后代节点都会被替换)
  - 不同类型的节点，产生不同的树结构(当根节点为不同类型的元素时，React 会拆卸原有的树并且建立起新的树)
  - 开发中，可以通过key属性标识哪些子元素在不同的渲染中可能是不变的
- 在遍历列表时，总是会提示一个警告，让我们加入一个key属性，当子元素拥有 key 时，React 使用 key 来匹配原有树上的子元素以及最新树上的子元素。
  - 在最后位置插入数据 -- 这种情况，有无key意义并不大
  - 在前面插入数据 -- 这种做法，在没有key的情况下，所有的li都需要进行修改
  - 在中间插入元素 -- 新增2014, key为2016元素仅仅进行位移，不需要进行任何的修改

```
<ul>
  <li key="2015">Duke</li>
  <li key="2016">Villanova</li>
</ul>

<ul>
  <li key="2015">Duke</li>
  <li key="2014">Connecticut</li>
  <li key="2016">Villanova</li>
</ul>
```

React的setState是同步的还是异步的？React18中是怎么样的?

- 在 React 中，可变状态通常保存在组件的 state 属性中，并且只能通过使用 setState()来更新
- React的setState是异步的 -- 不要指望在调用 `setState` 之后， `this.state` 会立即映射为新的值
- 在react18之前, 在setTimeout,Promise等中操作setState, 是同步操作
- 在react18之后, 在setTimeout,Promise等中操作setState,是异步操作(批处理)
  - 如果需要同步的处理怎么办呢? 需要执行特殊的 `flushSync` 操作
- 为什么要将setState设计成异步的

- 首先,若是将setState设计成同步的,在`componentDidMount`中请求多个网络请求时,会堵塞后面的网络请求

```
componentDidMount() {
  // 网络请求一 : this.setState
  // 网络请求二 : this.setState
  // 网络请求三 : this.setState
  // 如果this.setState设计成同步的,会堵塞后面的网络请求
}
```

- 一. 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**

```
// 在一个函数中有多个setState时,
this.setState({}) --> 先不会更新,而是会加入到队列(queue)中 (先进先出)
this.setState({}) --> 也加入到队列中
this.setState({}) --> 也加入到队列中
// 这里的三个setState会被合并到队列中去
// 在源码内部是通过do...while从队列中取出依次执行的
```

- 二: 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

什么是SCU优化？类组件和函数组件分别如何进行SCU的优化？

- shouldComponentUpdate — SCU ， React提供给我们的声明周期方法
- SCU优化就是 一种巧妙的技术,用来减少DOM操作次数,具体为当React元素没有更新时,不会去调用render()方法
- 可以通过`shouldComponentUpdate`来判断`this.state`中的值是否改变

```
shouldComponentUpdate(nextProps, nextState) {
  const {message, counter} = this.state
  if(message !== nextState.message || counter !== nextState.counter) {
    return true
  }
  return false
}
```

- React已经帮我们提供好SCU优化的操作
  - 类组件: 将class继承自`PureComponent`
  - 函数组件: 使用一个高阶组件`memo`

```
import {mome} from 'react'

const HomeFunc = mome(function(props) {
  console.log("函数式组件")
  return (
    <h4>函数式组件: {props.message}</h4>
  )
})

export default HomeFunc
```

什么是高阶组件？高阶组件在React开发中起到什么作用？

- 高阶函数: (满足一下调教之一) -- filter、map、reduce都是高阶函数
  - 接受一个或多个函数作为输入
  - 输出一个函数
- 高阶组件: Higher-Order Components，简称为 HOC
  - 高阶组件是参数为组件，返回值为新组件的函数 -- 就是传入一个组件,对这个组件进行一些功能的增强,在返回出来新的组件
  - 注意: 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
  - 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
- 高级组件应用的场景
  - props的增强
  - 利用高阶组件来共享Context
  - 渲染判断鉴权
  - 生命周期劫持
  - ....

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>
    )
  }
}
```

什么是React的严格模式，在开发中有什么作用?

严格模式

- StrictMode 是一个用来突出显示应用程序中潜在问题的工具：
- 与 Fragment 一样，StrictMode 不会渲染任何可见的 UI；
- 它为其后代元素触发额外的检查和警告；

严格模式作用

- 识别不安全的生命周期：
- 使用过时的ref API
- 检查意外的副作用 这个组件的constructor会被调用两次； 这是严格模式下故意进行的操作，让你来查看在这里写的一些逻辑代码被调用多次时，是否会产生一些副作用； 在生产环境中，是不会被调用两次的；
- 使用废弃的findDOMNode方法 在之前的React API中，可以通过findDOMNode来获取DOM，不过已经不推荐使用了，可以自行学习演练一下
- 检测过时的context API 早期的Context是通过static属性声明Context对象属性，通过getChildContext返回Context对象等方式来使用Context的

React中编写CSS的方式有哪些？各自有什么优缺点?

- 内联样式:
  - 优点:内联样式, 样式之间不会有冲突;可以动态获取当前state中的状态
  - 缺点:写法上都需要使用驼峰标识;某些样式没有提示;大量的样式, 代码混乱;某些样式无法编写(比如伪类/伪元素)

- 普通的css文件:
  - 优点:单独的css文件方便管理
  - 缺点:所有的样式都是全局的,样式之间会相互层叠掉
- css modules:
  - 优点: 样式文件需要修改成 .module.css/.module.less/.module.scss;引入后局部生效,样式直接不会相互影响
  - 缺点:引用的类名，不能使用连接符(.home-title)，在JavaScript中是不识别的;所有的className都必须使用{style.className} 的形式来编写;不方便动态来修改某些样式，依然需要使用内联样式的方式
- CSS in JS:通过JavaScript来为CSS赋予一些能力，包括类似于CSS预处理器一样的样式嵌套、函数定义、逻辑复用、动态修 改状态等等

styled-components有哪些技术特点？可以完成哪些功能?

- styled-components是最流行的CSS-in-JS库
- styled-components的本质是通过函数的调用，最终创建出一个组件:
  - 这个组件会被自动添加上一个不重复的class
  - styled-components会给该class添加相关的样式
- 支持类似于CSS预处理器一样的样式嵌套:
  - 支持直接子代选择器或后代选择器，并且直接编写样式
  - 可以通过&符号获取当前元素
  - 直接伪类选择器、伪元素等
- 支持传递props
- 支持样式的继承
- styled设置主题

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的核心思想是什么?

- Redux是JavaScript的状态容器，提供了可预测的状态管理
- Redux的核心理念:
  - store:用来存储状态
  - action:通过派发(dispatch)action来更新数据
  - reducer:纯函数,将传入的state和action结合起来生成一个新的state
- Redux的三大原则:
  - 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;

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；

redux如何进行文件，每个文件是什么作用?

- 将store、reducer、action、constants拆分成一个个单独文件
  - index.js文件:初始化store
  - reducer.js文件:初始化state,创建reducer函数
  - actionCreators.js文件:创建action的函数
  - constants.js文件:定义action中的type常量

什么是Redux Toolkit？核心API有哪些?

- configureStore
  - 包装createStore,同时提供简化的配置选项和良好的默认值
  - 自动组合单独的slice reducer
  - 可以添加任何中间件
    - 默认包含redux-thunk,并启用Redux-DevTools调试工具
- createSlice
  - 接受一个具有render函数的对象
  - 可以配置切片名称
  - 初始状态值
  - 自动 生成切片,并带有相应的actions
- createAsyncThunk
  - 接受一个动作类型字符和一个返回承诺的函数
  - 生成一个pending/fulfilled/rejected基于该承诺分配动作类型的Thunk

Redux原始的使用步骤。

- 先从react-redux中导入Providre包裹根组件
- 将导出的store绑定到Provider组件的store属性中
- 创建store,目录结构
  - actionCreator ---> 创建action对象
  - constant ---> 定义常量数据
  - reducer ---> 处理action对象,返回最新的state
  - index ---> 人口文件,创建store,使用中间件
- 组件中的使用方式
  - 定义函数 ---> mapStateToProps ---> 将store中的数据映射要组件的props中
  - 定义函数 ---> mapDispatchToProps ---> 将dispatch的操作映射到props中

    - 从react-redux中导入高阶组件对要导出的组件进行包裹,并把定义的函数传入connect函数
    - 组件触发相应的事件,dispatch相应的对象,store中的数据改变,组件重新渲染

Redux在RTK中使用步骤。

- 先从react-redux中导入Providre包裹根组件
- 将导出的store绑定到Provider组件的store属性中
- 创建store,目录结构:
    - index.js ---> 入口文件 ---> 创建和配置store ---> 主要是合并render
    - features ---> 要管理的数据模块
        - 使用createSliceAPI创建一个slice对象
        - name ---> 配置slice对象的名称
        - initialState ---> 定义初始值
        - reducer ---> 定义reduce函数的对象
        - 导出slice对象的actions---> 组件中使用或者自己内部使用,
        - 导出slice对象的reducer---> index文件合并reducer

Redux中有哪些Hooks？如何使用对应的Hooks?

- useSelector
    - 可以将redux中的数据映射到组件中
    - 第一个参数是一个回调函数
        - 函数参数是state
        - 返回一个对象,对象中包含state中的数据
    - 第二个参数是showEqual
        - 作用 ---> 只有当前获取的对应数据改变后,对应的组件才会重新渲染
- useDispatch
    - 没有参数
    - 返回一个diapatch函数 ---> 可以传入action对象

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。

React Router6路由创建过程?

- 安装react-router-dom
  - npm install react-router-dom
- 选择路由模式 BrowserRouter使用**history模式** / HashRouter使用**hash模式**

```
<HashRouter>
    <App />
</HashRouter>
```

- 通过Routes包裹所有的Route, 在其中匹配路由
- Route用于路径的匹配
  - path属性：用于设置匹配到的路径
  - element属性：设置匹配到路径后，渲染的组件 -- Router5.x使用的是component属性
- 路由跳转 -- Link组件 / NavLink组件
  - to属性 -- 用于设置跳转到的路径
- 路由重定向 -- Navigate
- Not Found页面配置 -- path="*"

```
<Link to="/home">首页</Link>
<Link to="/about">关于</Link>
<Routes>
    <Route path="/" element={<Navigate to="/home" />} />
    {/**Home页面*/}
    <Route path="/home" element={<Home />}>
      <Route path="/home" element={<Navigate to="/home/homebanner" />} />
      <Route path="/home/homebanner" element={<HomeBanner />} />
      <Route path="/home/homerecommend" element={<HomeRecommend />} />
    </Route>
    <Route path="/about" element={<About />}></Route>
    <Route path="*" element={<NotFount />}></Route>
</Routes>
```

React Router6如何进行路由配置？如何配置路由的懒加载?

路由配置 和 路由的懒加载

```
// 在单独的router/index.js文件中

// 路由懒加载/按需加载/异步加载 ?
// 这样暂时不会显示,因为是异步的要单独下载,需要加载一个loading动画React提供的Suspense组件
const Order = React.lazy(() => import("../page/Order"));
const User = React.lazy(() => import("../page/User"));

// 路由配置
const router = [
```

```
    {
        path: '/',
        element: <Navigate to="/home" />,
    },
    {
        path: "/home",
        element: <Home />,
        children: []
    }
]
export default router
```

这里使用了HashRouter，并且异步组件需要放在Suspense中

```
<HashRouter>
  <Suspense fallback={<h4>Loading~~~~</h4>}>
    <App />
  </Suspense>
</HashRouter>
```

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>
    )
  }
}
```

React Hooks

什么是Hooks？函数式组件和类组件有什么区别?

- **Hook指的类似于useState、useEffect这样的函数, Hooks是对这类函数的统称**
- 函数式组件与类组件的优缺点
  - 类组件可以定义自己的state,并且可以保存自己内部的状态
  - 函数式组件不能定义自己的状态的, 因为函数每次调用都会产生新的临时变量
  - 类组件有自己的生命周期 -- 而函数式组件没有
  - 类组件在状态改变是会重新执行render函数
  - 函数式组件时不会重新渲染的, 如果重新渲染, 整个函数会被重新执行, 相应的状态也会被重新赋值
- 同时类组件也有自己的缺点
  - 随着业务的增多,类组件会变得越来越复杂
  - 复用其中的状态也会很艰难,有时需要通过一些高阶组件
- **Hooks**可以让我们在不编写class的情况下使用state以及其他的React特性
  - Hook只能在函数组件中使用，不能在类组件
  - 通过Hook可以在函数式组件中 定义自己的状态 完成类似于class组件中的生命周期功能

类组件实现计数器

```
// 类组件实现计数器
import React, { PureComponent } from 'react'

export class CounterClass extends PureComponent {
  constructor() {
    super()
    this.state = {
      counter: 100
    }
  }
  subCounter() {
    this.setState({counter: this.state.counter - 1})
  }
  addCounter() {
    this.setState({counter: this.state.counter + 1})
  }
  render() {
    const { counter } = this.state
    return (
      <div>
        类组件实现计数器
        <h2>counter: {counter}</h2>
        <button onClick={() => this.subCounter(1)}>-1</button>
        <button onClick={() => this.addCounter(1)}>+1</button>
      </div>
    )
  }
}
```

```
export default CounterClass
```

Hooks实现计数器

```
// Hooks实现计数器
import React, { memo, useState } from 'react'
const CounterHooks = memo(() => {
  let [counter, setCounter] = useState(100)
  return (
    <div>
      Hooks
      <h2>counter: {counter}</h2>
      <button onClick={e => setCounter(counter - 1)}>-1</button>
      <button onClick={e => setCounter(counter + 1)}>+1</button>
    </div>
  )
})

export default CounterHooks
```

常见的Hooks，以及说明它们的作用。

- useState
  - 在函数组件中用来保存组件的状态，即定义响应式的变量
  - 参数
    - 定义数据的默认值
  - 返回一个数组
    - 第一项是：定义数据的变量, 可以在jsx中使用该变量
    - 第一项是：一个函数, 调用这个函数可以设置新的数据, 函数式组件会重新渲染
- useEffect
  - 在函数组件中可以让我实现在class组件中类似于生命周期的功能
  - 参数：
    - 传入一个回调函数,这个函数会在更新完DOM后执行
    - 传入的回调函数可以返回一个函数,这个函数会在组件卸载时被调用
- useContext
  - 通过这个hook可以直接获取到某个context的值
  - 避免使用多个Context共享时存在的大量嵌套
  - 参数
    - 传入context对象
  - 返回值
    - 返回定义在context中定义的数据

- useReducer
  - 适合处理逻辑比较复杂的state, 可以通过useReducer进行拆分
  - 本次修改的state需要依赖之前的state
  - 参数
    - 第一个参数是类似redux中reducer的纯函数,返回新的state
    - 第二个参数是我们想要定义的数据
  - 返回值
    - 数组类型
    - 第一个元素是我们定义的state
    - 第二个参数是函数dispatch,调用时可以传递一个对象action,这个action会传递到前面定义reducer
- useCallback
  - 参数
    - 第一个参数是回调函数
    - 第二个参数是依赖，为数组类型
  - 返回值
    - 是一个函数, 如果在依赖的数据没有变化的情况下, 返回函数同一个对象
- useMemo
  - 参数
    - 第一个参数是回调函数
    - 第二个参数是依赖
  - 返回值
    - 是一个对象, 如果在依赖的数据没有变化的情况下, 返回同一个对象
- useRef
  - 引入DOM元素或者组件对象
  - 也可以用来保存一个数据
  - 返回值
    - 返回一个对象, 这个对象在整个生命周期中可以保持不变
- useImperativeHandle
- uselayoutEffect
  - useEffect会在渲染的内容更新到DOM上后执行,不会阻塞DOM的更新
  - useLayoutEffect会在渲染的内容更新到DOM上之前执行,会阻塞Dom的更新

useEffect有哪些使用方式?

- 可以在同一个函数组件中多次使用useEffect
  - 好处: 可以将不同的代码逻辑拆分,不会像生命周期函数一样编写到一块
- 清除effect
  - 在useEffect中传入的回调函数可以返回一个函数

    - 这个函数会在组件卸载时被调用
- 性能优化
    - 第二个参数传入要依赖的数据, 为数组类型
    - 当依赖的数据没有发生变化时, useEffect中的回调函数不会执行（类似Vue中的watch）

useMemo和useCallback有什么区别?

- useMemo
    - 返回值是一个对象
    - 可以实现 useCallback 一样的功能
- useCallback
    - 返回值是一个函数

React18新增哪些Hooks，这些Hooks有什么作用?

- useId
    - 用于生产横跨服务端和客户端的稳定的唯一的ID的同时避免hydration不匹配的hook
- useTransition
    - 降低某部分任务的更新优先级,先完成优先级较高的任务后执行
- useDeferredValue
    - 接受一个值,并返回该值的 新副本,该副本将在优先级较高的任务完成后执行

什么是SPA？什么是SSR？什么是同构应用？什么是hydration?

- SPA
    - Single Page Application, 单页面应用
    - 一种Web应用程序,它只需要将单个页面夹加载到浏览器中
    - 特点
        - 一旦页面加载完成,SPA不会因为用户的操作而进行页面的重新加载或跳转
        - 利用路由机制实现页面内容的变换
    - 优点
        - 用户体验好、快、内容的改变不需要重新加载整个页面，避免了不必要的跳转和重复渲染，SPA 相对服务器压力小
        - 前后端职责分离，架构清晰，前后端进行交互逻辑，后端负责数据处理
    - 缺点
        - 初次加载耗时多：为实现单页 Web 的应用功能及显示效果，需要在加载页面的时候将 JavaScript、CSS统一加载，部分页面按需加载
        - SEO难度较大：由于所有的内容都在一个页面中动态替换显示，所以在SEO上其有着天然的弱势。

        - 如果你使用了AJAX进行局部刷新，浏览过的内容就不能用后退按钮重现了。
- SSR
    - Server Side Rendering, 服务端渲染的简称
    - 在服务器端生成完整的HTML页面结构,返回给浏览器进行渲染
    - 特点
        - 在服务端生成html网页的dom元素
        - 客户端（浏览器）只负责显示dom元素内容
    - 优点
        - 有利于SEO，网站通过href的url将搜索引擎直接引到服务端，服务端提供优质的网页内容给搜索引擎。
    - 缺点
        - 需要消耗一定的服务器资源
- 同构应用
    - 一套代码既可以在服务端运行又可以在客户端运行,我们称为同构应用
    - 同构是一种SSR的形态,是现代SSR的一种表现形式
- hydration
    - SSR引申出来的概念
    - SSR ----> 服务端将生成好的HTML页面发送浏览器进行展示,这其中是没有js逻辑的,比如事件的绑定,也就是不能和用户进行交互的
    - 在服务端渲染的时候,UI框架(React/Vue)会记录SSR渲染的页面的内部特征,然后将这个特只能特征映射到我们要在浏览器展示的页面上,它会让我们的页面具有交互性,这个过程称之为Hydration