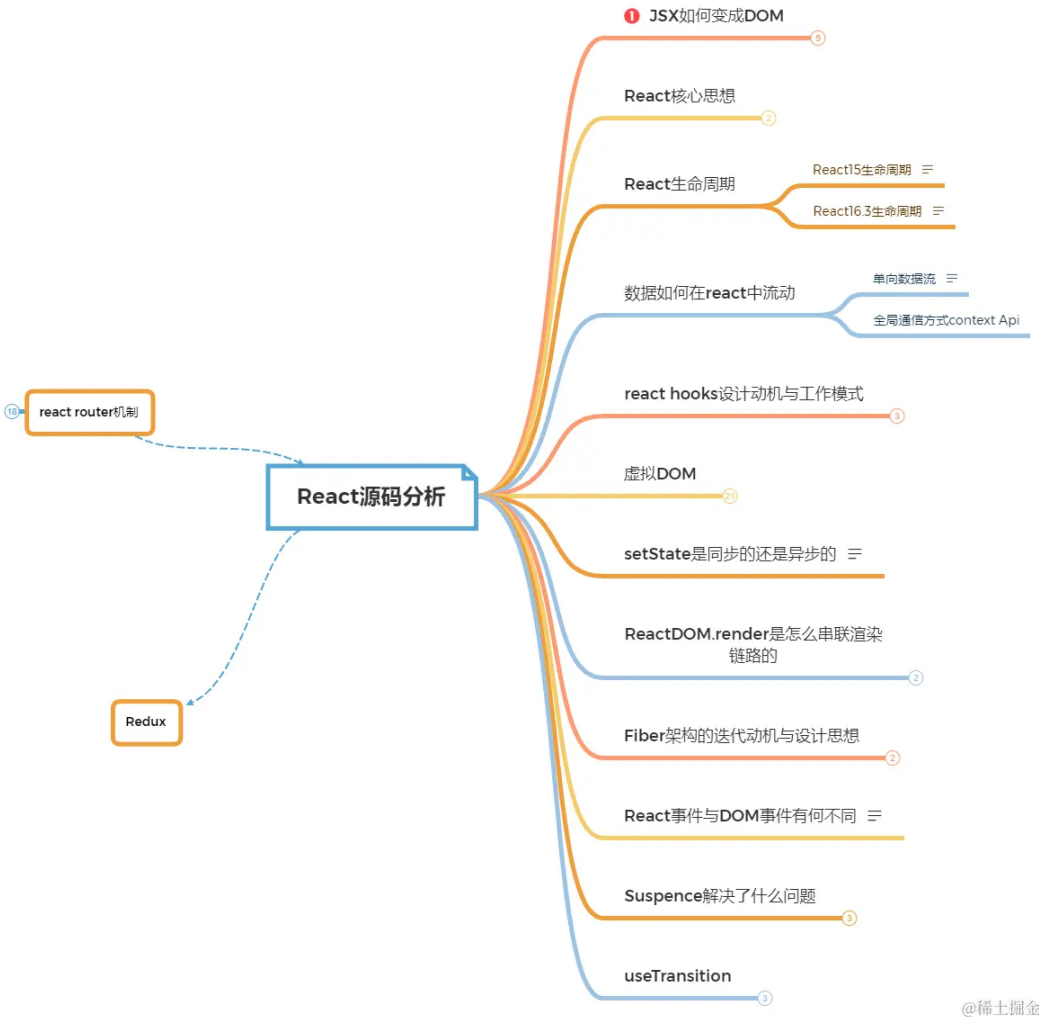
一文带你梳理React面试题（源码分析）

源码分析



一、React18有哪些更新？

juejin.cn/post/709403...

1. setState自动批处理

在react17中，只有react事件会进行批处理，原生js事件、promise，setTimeout、setInterval不会

react18，将所有事件都进行批处理，即多次setState会被合并为1次执行，提高了性能，在数据层，将多个状态更新合并成一次处理（在视图层，将多次渲染合并成一次渲染）

1. 引入了新的root API，支持new concurrent renderer(并发模式的渲染)

```
import React from "react"
import ReactDOM from "react-dom"
import App from "./App"

const root = document.getElementById("root")
ReactDOM.render(<App/>,root)


ReactDOM.unmountComponentAtNode(root)


import React from "react"
import ReactDOM from "react-dom/client"
import App from "./App"
const root = document.getElementById("root")
ReactDOM.createRoot(root).render(<App/>)


root.unmount()
```

3\. 去掉了对IE浏览器的支持，react18引入的新特性全部基于现代浏览器，如需支持需要退回到react17版本 4. flushSync

批量更新是一个破坏性的更新，如果想退出批量更新，可以使用flushSync

```
import React,{useState} from "react"
import {flushSync} from "react-dom"
```

```
const App=()=>{
  const [count,setCount]=useState(0)
  const [count2,setCount2]=useState(0)

  return (
    <div className="App">
      <button onClick={()=>{
        // 第一次更新
        flushSync(()=>{
          setCount(count=>count+1)
        })
        // 第二次更新
        flushSync(()=>{
          setCount2(count2=>count2+1)
        })
      })>点击</button>
      <span>count:{count}</span>
      <span>count2:{count2}</span>
    </div>
  )
}
export default App
```

1. react组件返回值更新

- 在react17中，返回空组件只能返回null，显式返回undefined会报错
- 在react18中，支持null和undefined返回

1. strict mode更新

当你使用严格模式时，React会对每个组件返回两次渲染，以便你观察一些意想不到的结果,在react17中去掉了一次渲染的控制台日志，以便让日志容易阅读。react18取消了这个限制，第二次渲染会以浅灰色出现在控制台日志

1. Suspense不再需要fallback捕获
2. 支持useId

在服务器和客户端生成相同的唯一一个id，避免hydrating的不兼容

1. useSyncExternalStore

用于解决外部数据撕裂问题

1. useInsertionEffect

这个hooks只建议在css in js库中使用，这个hooks执行时机在DOM生成之后，useLayoutEffect执行之前，它的工作原理大致与useLayoutEffect相同，此时无法访问DOM节点的引用，一般用于提前注入脚本

1. **Concurrent Mode**

并发模式不是一个功能，而是一个底层设计。

它可以帮助应用保持响应，根据用户的设备性能和网速进行调整，它通过渲染可中断来修复阻塞渲染机制。在**concurrent模式**中，React可以同时更新多个状态

区别就是使**同步不可中断更新**变成了**异步可中断更新**

useDeferredValue和startTransition用来标记一次非紧急更新

二、React的设计思想

- **组件化**

每个组件都符合开放-封闭原则，封闭是针对渲染工作流来说的，指的是组件内部的状态都由自身维护，只处理内部的渲染逻辑。开放是针对组件通信来说的，指的是不同组件可以通过props（单项数据流）进行数据交互

- **数据驱动视图**

UI=f(data)

通过上面这个公式得出，如果要渲染界面，不应该直接操作DOM，而是通过修改数据(state或prop)，数据驱动视图更新

- **虚拟DOM**

由浏览器的渲染流水线可知，DOM操作是一个昂贵的操作，很耗性能，因此产生了虚拟DOM。虚拟DOM是对真实DOM的映射，React通过新旧虚拟DOM对比，得到需要更新

的部分，实现数据的增量更新

React设计模式

三、JSX是什么，它和JS有什么区别

JSX是react的语法糖，它允许在html中写JS，它不能被浏览器直接识别，需要通过webpack、babel之类的编译工具转换为JS执行

JSX与JS的区别：

1. JS可以被打包工具直接编译，不需要额外转换，jsx需要通过babel编译，它是React.createElement的语法糖，使用jsx等价于React.createElement
2. jsx是js的语法扩展，允许在html中写JS；JS是原生写法，需要通过script标签引入

为什么在文件中没有使用react，也要在文件顶部import React from “react”

只要使用了jsx，就需要引用react，因为jsx本质就是React.createElement

> 注意，在React 17RC版本后，jsx不一定会被转换为React.createElement了

以下代码

```
function App(){
    return <h1>hello,lyllovelemon</h1>
}
```

react17将会通过编译器babel/typescript转换为

```
import {jsx as _jsx} from 'react/jsx-runtime';

function App() {
  return _jsx('h1', { children: 'hello,lyllovelemon' });
}
```

此时就不需要通过import React就能使用jsx了（用react hooks还是需要导入React）

为什么React自定义组件首字母要大写

jsx通过babel转义时，调用了React.createElement函数，它接收三个参数，分别是type元素类型，props元素属性，children子元素。

如下图所示，从jsx到真实DOM需要经历_jsx→虚拟DOM→真实DOM_。如果组件首字母为小写，它会被当成字符串进行传递，在创建虚拟DOM的时候，就会把它当成一个html标签，而html没有app这个标签，就会报错。组件首字母为大写，它会当成一个变量进行传递，React知道它是个自定义组件就不会报错了

```
<app>lyllovelemon</app>

React.createElement("app",null,"lyllovelemon")

<App>lyllovelemon</App>

React.createElement(App,null,lyllovelemon)
```

React组件为什么不能返回多个元素

这个问题也可以理解为React组件为什么只能有一个根元素，原因：

1. React组件最后会编译为render函数，函数的返回值只能是1个，如果不用单独的根节点包裹，就会并列返回多个值，这在js中是不允许的

```
class App extends React.Component{
  render(){
    return(
    <div>
     <h1 className="title">lyllovelemon</h1>
      <span>内容</span>
    </div>
  )
}
```

```
class App extends React.Component{
  render(){
    return React.createElement('div',null,[
      React.createElement('h1',{className:'title'},'lyllovele
mon'),
      React.createElement('span'),null,'内容'
      ])
  }
}
```

2\. react的虚拟DOM是一个树状结构，树的根节点只能是1个，如果有多个根节点，无法确认是在哪棵树上进行更新

vue的根节点为什么只有一个也是同样的原因

React组件怎样可以返回多个组件

- 使用HOC（高阶函数）
- 使用React.Fragment,可以让你将元素列表加到一个分组中，而且不会创建额外的节点（类似vue的template)

```
renderList(){
  this.state.list.map((item,key)=>{
    return (<React.Fragment>
      <tr key={item.id}>
        <td>{item.name}</td>
        <td>{item.age}</td>
        <td>{item.address}</td>
      </tr>
    </React.Fragment>)
  })
}
```

3\. 使用数组返回

```
renderList(){
  this.state.list.map((item,key)=>{
    return [
      <tr key={item.id}>
        <td>{item.name}</td>
        <td>{item.age}</td>
        <td>{item.address}</td>
      </tr>
    ]
  })
}
```

React中元素和组件的区别

react组件有类组件、函数组件

react元素是通过jsx创建的

```
const element = <div className="element">我是元素</div>
```

四、简述React的生命周期

生命周期指的是组件实例从创建到销毁的流程，函数组件没有生命周期，只有类组件才有，因为只有class组件会创建组件实例

组件的生命周期可以分为**挂载、更新、卸载**阶段，下面说的是React16.3版本后的生命周期，之前版本的请自行查阅

挂载

constructor 初始化阶段，可以进行state和props的初始化

static getDerivedStateFromProps 静态方法，不能获取this

render 创建虚拟DOM的阶段

componentDidMount 第一次渲染后调用，挂载到页面生成真实DOM，可以访问DOM，进行异步请求和定时器、消息订阅

更新

当组件的props或state变化会触发更新

static getDerivedStateFromProps

shouldComponentUpdate 返回一个布尔值，默认返回true，可以通过这个生命周期钩子进行**性能优化**，确认不需要更新组件时调用

```
shouldComponentUpdate(nextProps,nextState){

  if(this.props === nextProps && this.state === nextState){
    return false
  }

  return true
}
```

render 更新虚拟DOM

getSnapShotBeforeUpdate 获取更新前的状态

componentDidUpdate 在组件完成更新后调用，更新挂载后生成真实DOM

卸载

componentWillUnmount 组件从DOM中被移除的时候调用，通常在这个阶段清除副作用，比如定时器、事件监听等

错误捕获

static getDerivedStateFromError 在errorBoundary中使用

componentDidCatch

render是class组件中唯一必须实现的方法

五、React事件机制

什么是合成事件

React基于浏览器的事件机制实现了一套自身的事件机制，它符合W3C规范，包括事件触发、事件冒泡、事件捕获、事件合成和事件派发等

React事件的设计动机(作用)：

- **在底层磨平不同浏览器的差异，React实现了统一的事件机制，我们不再需要处理浏览器事件机制方面的兼容问题，在上层面向开发者暴露稳定、统一的、与原生事件相同的事件接口**
- **React把握了事件机制的主动权，实现了对所有事件的中心化管控**
- **React引入事件池避免垃圾回收，在事件池中获取或释放事件对象，避免频繁的创建和销毁**

React事件机制和原生DOM事件流有什么区别

虽然合成事件不是原生DOM事件，但它包含了原生DOM事件的引用，可以通过e.nativeEvent访问

DOM事件流是怎么工作的，一个页面往往会绑定多个事件，页面接收事件的顺序叫事件流

W3C标准事件的传播过程：

1. 事件捕获
2. 处于目标
3. 事件冒泡

常用的事件处理性能优化手段：**事件委托**

把多个子元素同一类型的监听函数合并到父元素上，通过一个函数监听的行为叫事件委托

我们写的React事件是绑定在DOM上吗，如果不是绑定在哪里

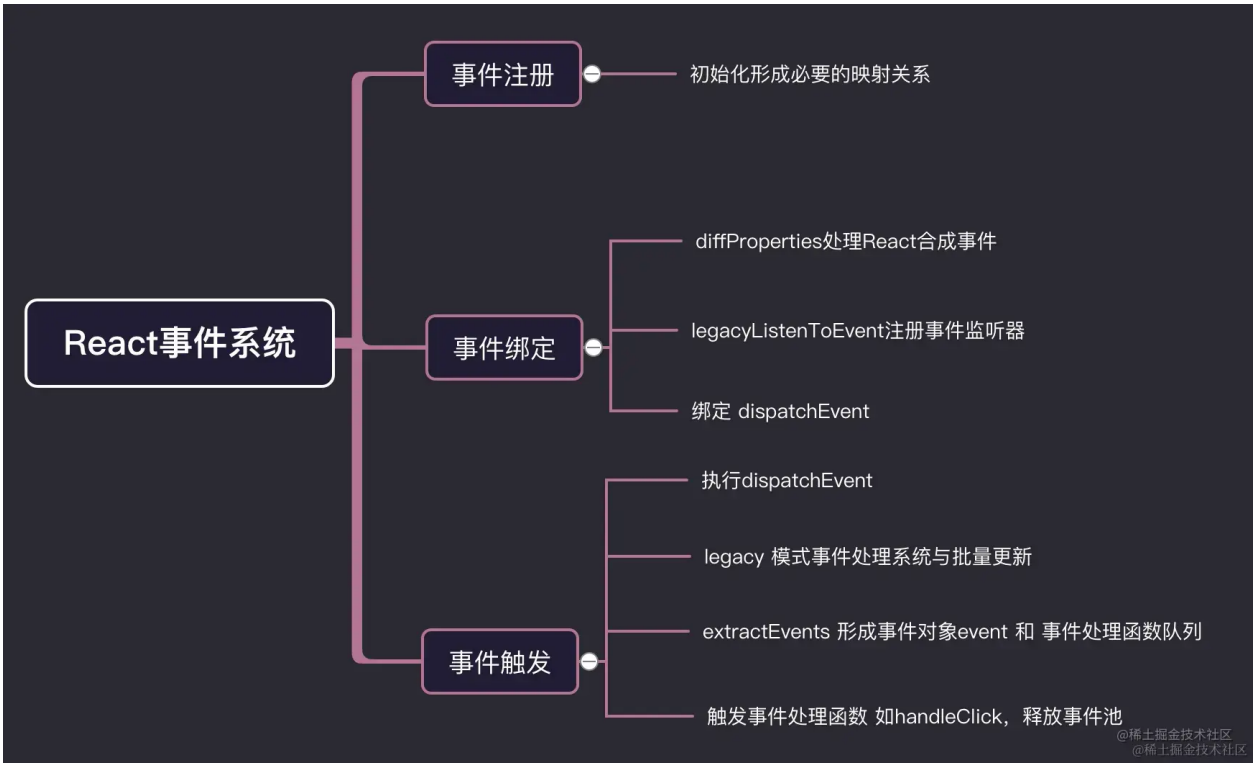
React16的事件绑定在document上， React17以后事件绑定在container上,**ReactDOM.render(app,container)**

React事件机制总结如下：

事件绑定 事件触发

- **React所有的事件绑定在container上**(react17以后),而不是绑定在DOM元素上（作用：减少内存开销，所有的事件处理都在container上，其他节点没有绑定事件）
- React自身实现了一套冒泡机制，不能通过return false阻止冒泡

- React通过**SytheticEvent**实现了**事件合成**



React实现事件绑定的过程

1.建立合成事件与原生事件的对应关系

registrationNameModule, 它建立了React事件到plugin的映射，它包含React支持的所有事件的类型，用于判断一个组件的prop是否是事件类型

```
{
   onBlur:SimpleEventPlugin,
   onClick:SimpleEventPlugin,
   onClickCapture:SimpleEventPlugin,
   onChange:ChangeEventPlugin,
   onChangeCapture:ChangeEventPlugin,
   onMouseEnter:EnterLeaveEventPlugin,
   onMouseLeave:EnterLeaveEventPlugin,
   ...
```

```
}
```

registrationNameDependencies， 这个对象记录了React事件到原生事件的映射

```
{
  onBlur: ['blur'],
  onClick: ['click'],
  onClickCapture: ['click'],
  onChange: ['blur', 'change', 'click', 'focus', 'input', 'ke
ydown', 'keyup', 'selectionchange'],
  onMouseEnter: ['mouseout', 'mouseover'],
  onMouseLeave: ['mouseout', 'mouseover'],
}
```

plugins对象, 记录了所有注册的插件列表

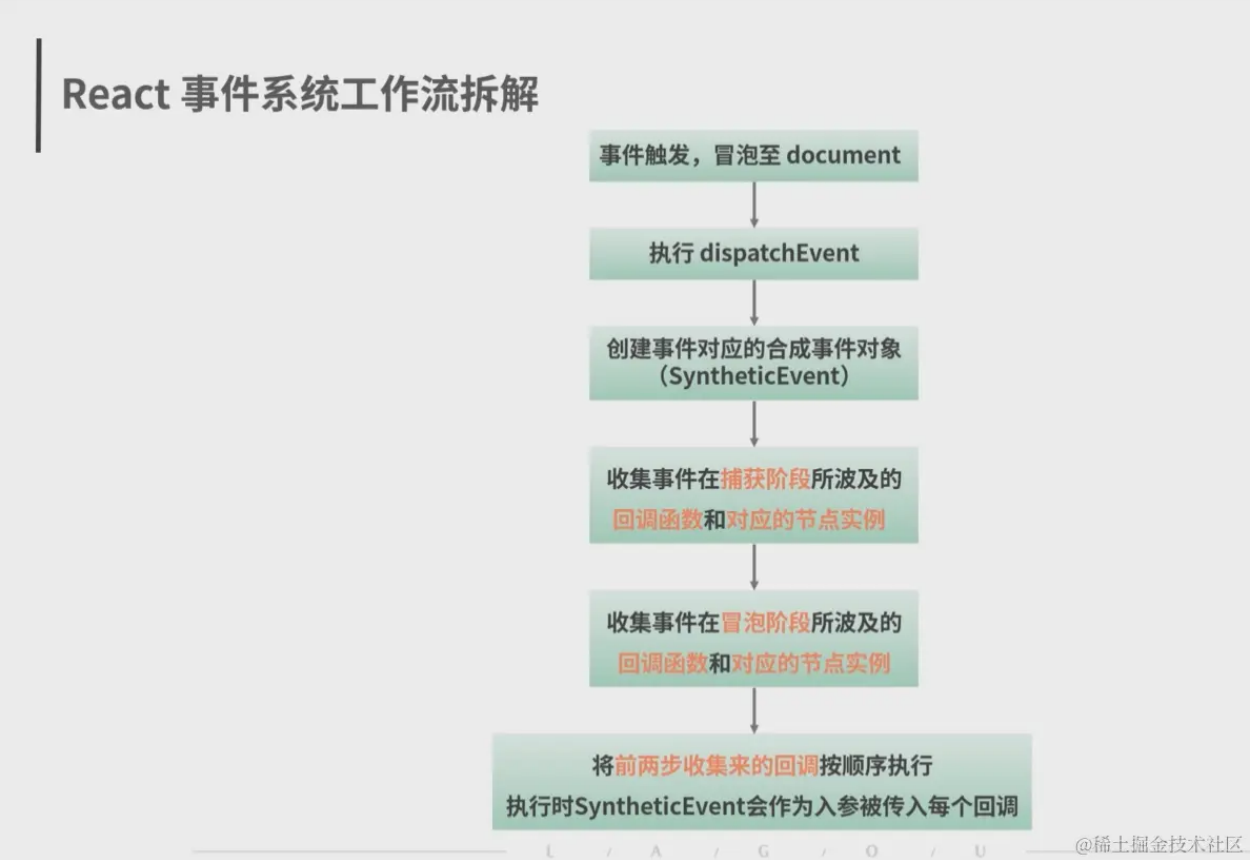
```
plugins = [LegacySimpleEventPlugin, LegacyEnterLeaveEventPlug
in, ...]
```

为什么针对同一个事件，即使可能存在多次回调，document（container）也只需要注册一次监听

因为React注册到document(container)上的并不是一个某个DOM节点具体的回调逻辑，而是一个统一的事件分发函数dispatchEvent - > 事件委托思想

dispatchEvent是怎么实现事件分发的

事件触发的本质是对dispatchEvent函数的调用



React事件处理为什么要手动绑定this

react组件会被编译为React.createElement,在createElement中，它的this丢失了，并不是由组件实例调用的，因此需要手动绑定this

为什么不能通过return false阻止事件的默认行为

因为React基于浏览器的事件机制实现了一套自己的事件机制，和原生DOM事件不同，它采用了事件委托的思想，通过dispatch统一分发事件处理函数

React怎么阻止事件冒泡

- 阻止合成事件的冒泡用e.stopPropagation()
- 阻止合成事件和最外层document事件冒泡，使用e.nativeEvent.stopImmediatePropogation()
- 阻止合成事件和除了最外层document事件冒泡，通过判断e.target避免

```
document.body.addEventListener('click',e=>{
  if(e.target && e.target.matches('div.stop')){
```

```
      return
    }
    this.setState({active:false})
  })
```

HOC和hooks的区别

useEffect和useLayoutEffect区别

React性能优化手段

1. shouldComponentUpdate
2. memo
3. getDerviedStateFromProps
4. 使用Fragment
5. v-for使用正确的key
6. 拆分尽可能小的可复用组件，ErrorBoundary
7. 使用React.lazy和React.Suspense延迟加载不需要立马使用的组件

六、常用组件

错误边界

React部分组件的错误不应该导致整个应用崩溃，为了解决这个问题，React16引入了错误边界

使用方法：

React组件在内部定义了getDerivedStateFromError或者componentDidCatch，它就是一个错误边界。getDerviedStateFromError和componentDidCatch的区别是前者展示降级UI，后者记录具体的错误信息，它只能用于class组件

```
import React from "react"
class ErrorBoundary extends React.Component{
  constructor(props){
    super(props)
    this.state={
```

```
      hasError:false
    }
  }
  staic getDerivedStateFromError(){
    return { hasError:true}
  }
  componentDidCatch(err,info){
    console.error(err,info)
  }
  render(){
    if(this.state.hasError){
      return <div>Oops,err</div>
    }
    return this.props.children
  }
}


import React from "react"
import ErrorBoundary from "./components/ErrorBoundary"
import ComponentA from "./components/ComponentA"
export class App extends React.Component{
  render(){
    return (
      <ErrorBoundary>
        <ComponentA></ComponentA>
      </ErrorBoundary>
    )
  }
}
```

错误边界无法捕获自身的错误，也无法捕获事件处理、异步代码(setTimeout、requestAnimationFrame)、服务端渲染的错误

Portal

Portal提供了让子组件渲染在除了父组件之外的DOM节点的方式,它接收两个参数，第一个是需要渲染的React元素，第二个是渲染的地方(DOM元素)

```
ReactDOM.createPortal(child,container)
```

用途：弹窗、提示框等

Fragment

Fragment提供了一种将子列表分组又不产生额外DOM节点的方法

Context

常规的组件数据传递是使用props，当一个嵌套组件向另一个嵌套组件传递数据时，props会被传递很多层，很多不需要用到props的组件也引入了数据，会造成数据来源不清晰，多余的变量定义等问题，Context提供了一种跨层级组件数据传递的方法

```
const ThemeContext = React.createContext('light')
class App extends React.Component {
  render(){
    return(
      <ThemeContext.Provider value="dark">
        <ToolBar/>
      </ThemeContext>
    )
  }
}

function ToolBar(){
  return <div>
   <ThemeButton/>
  </div>
}

class ThemeButton extends React.Component {
  static contextType = ThemeContext
  render(){
```

```
    return <Button theme={this.context}></Button>
  }
}
```

Suspense

Suspense使组件允许在某些操作结束后再进行渲染，比如接口请求,一般与React.lazy一起使用

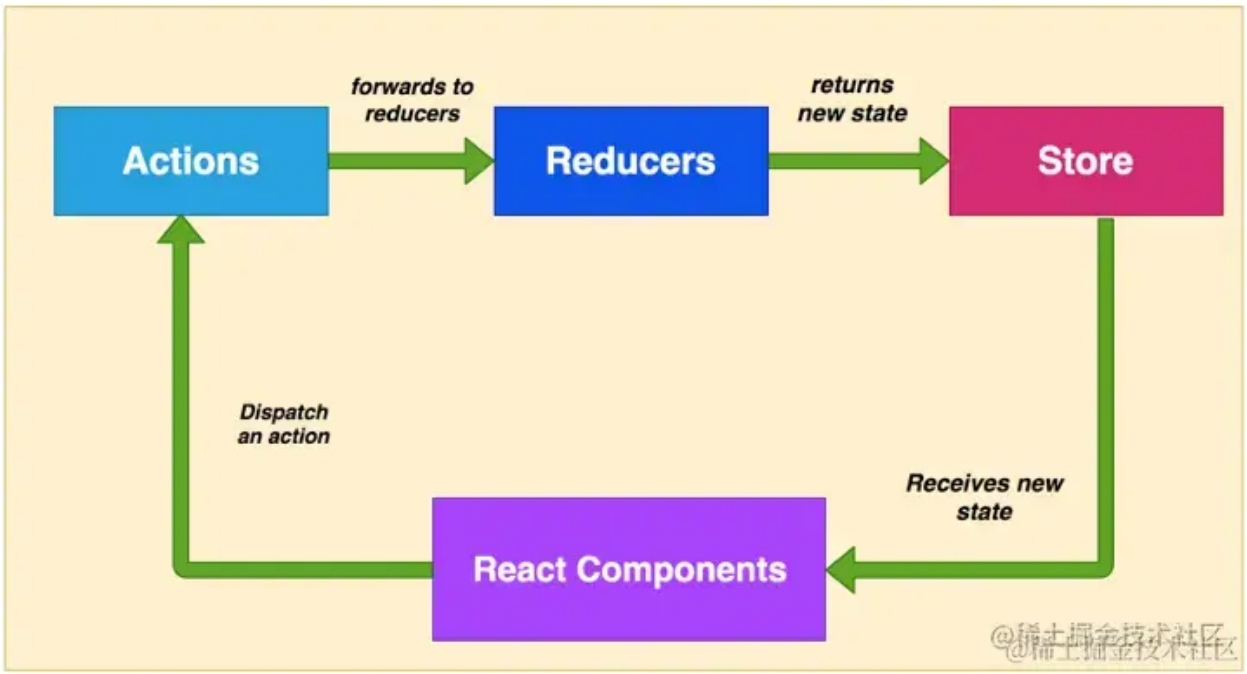
Transition

Transition是React18引入的一个并发特性，允许操作被中断，避免回到可见内容的Suspense降级方案

七、Redux工作原理

Redux是一个状态管理库，使用场景：

- 跨层级组件数据共享与通信
- 一些需要持久化的全局数据，比如用户登录信息



Redux工作原理

使用单例模式实现

Store 一个全局状态管理对象

Reducer 一个纯函数，根据旧state和props更新新state

Action 改变状态的唯一方式是dispatch action

八、React-Router工作原理

为什么需要前端路由

1. 早期：一个页面对应一个路由，路由跳转导致页面刷新，用户体验差
2. ajax的出现使得不刷新页面也可以更新页面内容，出现了_SPA_（单页应用）。_SPA_不能记住用户操作，只有一个页面对URL做映射，SEO不友好
3. 前端路由帮助我们在仅有一个页面时记住用户进行了哪些操作

前端路由解决了什么问题

1. 当用户刷新页面，浏览器会根据当前URL对资源进行重定向(发起请求)
2. 单页面对服务端来说就是一套资源，怎么做到不同的URL映射不同的视图内容
3. 拦截用户的刷新操作，避免不必要的资源请求；感知URL的变化

react-router-dom有哪些组件

HashRouter/BrowserRouter 路由器

Route 路由匹配

Link 链接，在html中是个锚点

NavLink 当前活动链接

Switch 路由跳转

Redirect 路由重定向

```
<Link to="/home">Home</Link>
<NavLink to="/abount" activeClassName="active">About</NavLink
>
```

```
<Redirect to="/dashboard">Dashboard</Redirect>
```

React Router核心能力：**跳转**

路由负责定义路径和组件的映射关系

导航负责触发路由的改变

路由器根据Route定义的映射关系为新的路径匹配对应的逻辑

BrowserRouter使用的**HTML5**的**history api**实现路由跳转

HashRouter使用URL的**hash属性**控制路由跳转

前端通用路由解决方案

- hash模式

> 改变URL以#分割的路径字符串，让页面感知路由变化的一种模式,通过_hashchange_事件触发

- history模式

> 通过浏览器的history api实现,通过_popState_事件触发

九、数据如何在React组件中流动

React组件通信

react组件通信方式有哪些

组件通信的方式有很多种，可以分为以下几种：

1. 父组件向子组件通信
2. 子组件向父组件通信
3. 兄弟组件通信
4. 父组件向后代组件通信
5. 无关组件通信

父组件向子组件通信

- **props传递**，利用React单向数据流的思想，通过props传递

```
function Child(props){
  return (
    <div className="child">
     <input value={props.name} type="text"/>
    </div>
  )
}
const Parent = <Child name="我是儿子"/>
```

子组件向父组件通信

- **回调函数**

父组件向子组件传递一个函数，通过函数回调，拿到子组件传过来的值

```
import React from "react"
class Parent extends React.Component{
  constructor(){
    super()
    this.state={
      price:0
    }
  }
  getPrice(val){
    this.setState({
      price:val
    })
  }
  render(){
    return (<div>
      <span className="label">价格:</span>
      <span className="value">{this.state.price}</span>
      <Child getPrice={this.getPrice.bind(this)}/>
    </div>)
```

```
  }
}

class Child extends React.Component{
  getItemPrice(e){
    this.props.getPrice(e)
  }
  render(){
    return (
      <div>
        <button onClick={this.getItemPrice.bind(this)}>廓形大衣
</button>
        <button onClick={this.getItemPrice.bind(this)>牛仔裤</
button>
      </div>
    )
  }
}
```

- **事件冒泡**

点击子组件的button按钮，事件会冒泡到父组件上

```
const Child = () => {
    return <button>点击</button>
}

const Parent = () => {
    const sayName = name => {
        console.log(name)
    }
    return (
        <div onClick={() => sayName('lyllovelemon')}>
            <Child />
        </div>
```

```
    )
}

export default Parent
```

- **Ref**

```
import React from "react"
class Parent extends React.Component{
  constructor(props){
    super(props)
    this.myRef=React.createRef()
  }
  componentDidMount(){
    this.myRef.current.changeVal('lyllovelemon')
  }
}
class Child extends React.Component{
  constructor(props){
    super(props)
  }
  changeVal(name){
    console.log(name)
  }
  render(){
    return (<div></div>)
  }
}
```

兄弟组件通信

实际上就是通过父组件中转数据的，子组件a传递给父组件，父组件再传递给子组件b

```
import React from "react"
class Parent extends React.Component{
  constructor(props){
    super(props)
    this.state={
      count:0
    }
  }
  increment(){
    this.setState({
      count:this.state.count+1
    })
  }
  render(){
    return (
      <div>
        <ChildOne count={this.state.count} />
        <ChildTwo onClick={this.increment} />
      </div>
    )
  }
}
```

父组件向后代组件通信

Context

```
import React from "react"
const PriceContext = React.createContext("price")
export default class Parent extends React.Component{
  constructor(props){
    super(props)
  }
  render(){
    return (
```

```
      <PriceContext.Provider value={200}>
      </PriceContext>
    )
  }
}
class Child extends React.Component{
  ...
}
class SubChild extends React.Component{
  constructor(props){
    super(props)
  }
  render(){
    return (
      <PriceContext.Consumer>
        { price=> <div>price:{price}</div> }
      </PriceContext.Consumner>
    )
  }
}
```

HOC

Redux

ref，useRef，forwardRef，useImperativeHandle

十、React Hooks

React hooks解决了什么问题

在React16.8以前，常用的组件写法有class组件和function组件

```
class Demo extends React.Component{
    constructor(props){
        super(props)
        this.state={
```

```
            name:'lyllovelemon'
        }
    }
    changeName(){
        this.setState({
            name:'lylmusiclover'
        })
    }
    render(){
        return (
            <div>
                {this.state.name}
                <button onClick={this.changeName.bind(this)}>
换名</button>
            </div>
        )
    }
}

function Demo2({name,setName}){
    return <div>
        {name}
        <button onClick={()=>setName('哈哈哈')}>换名</button>
    </div>
}


function App() {
    const [name,setName]=useState('lyl')
  return (
    <div className="App">
        <Demo/>
        <Demo2 setName={setName} name={name}/>
    </div>
  )
```

```
}
```

函数组件与类组件的区别:

1. 类组件需要声明constructor，函数组件不需要
2. 类组件需要手动绑定this，函数组件不需要
3. 类组件有生命周期钩子，函数组件没有
4. 类组件可以定义并维护自己的state，属于有状态组件，函数组件是无状态组件
5. 类组件需要继承class，函数组件不需要
6. 类组件使用的是面向对象的方法，封装：组件属性和方法都封装在组件内部 继承:通过extends React.Component继承;函数组件使用的是函数式编程思想

why React hooks

优点：

1. 告别难以理解的class组件
2. 解决业务逻辑难以拆分的问题
3. 使状态逻辑复用变的简单可行
4. 函数组件从设计理念来看，更适合react

局限性：

1. hooks还不能完整的为函数组件提供类组件的能力
2. 函数组件给了我们一定程度的自由，却也对开发者的水平提出了更高的要求
3. Hooks 在使用层面有着严格的规则约束

常用hooks

useState

juejin.cn/post/711893...

十一、SetState是同步还是异步的

setState是一个异步方法，但是在setTimeout/setInterval等定时器里逃脱了React对它的掌控，变成了同步方法

实现机制类似于vue的$nextTick和浏览器的事件循环机制，每个setState都会被react加入到任务队列，多次对同一个state使用setState只会返回最后一次的结果，因为它不是立刻就更新，而是先放在队列中，等时机成熟在执行批量更新。React18以后，使用了createRoot api后，所有setState都是异步批量执行的

十二、fiber架构

什么是fiber，fiber解决了什么问题

在React16以前，React更新是通过**树的深度优先遍历**完成的，遍历是不能中断的，当树的层级深就会产生栈的层级过深，页面渲染速度变慢的问题，为了解决这个问题引入了fiber，React fiber就是虚拟DOM，它是一个链表结构，返回了return、children、siblings，分别代表父fiber，子fiber和兄弟fiber，随时可中断

Fiber是纤程，比线程更精细，表示对渲染线程实现更精细的控制

应用目的

实现增量渲染，增量渲染指的是把一个渲染任务分解为多个渲染任务，而后将其分散到多个帧里。增量渲染是为了实现任务的可中断、可恢复，并按优先级处理任务，从而达到更顺滑的用户体验

Fiber的可中断、可恢复怎么实现的

_fiber_是协程，是比线程更小的单元，可以被人为中断和恢复，当react更新时间超过1帧时，会产生视觉卡顿的效果，因此我们可以通过fiber把浏览器渲染过程分段执行，每执行一会就让出主线程控制权，执行优先级更高的任务

fiber是一个链表结构，它有三个指针，分别记录了当前节点的下一个兄弟节点，子节点，父节点。当遍历中断时，它是可以恢复的，只需要保留当前节点的索引，就能根据索引找到对应的节点

Fiber更新机制

初始化

1. 创建fiberRoot（React根元素）和rootFiber(通过ReactDOM.render或者ReactDOM.createRoot创建出来的)
2. 进入beginWork

workInProgress:正在内存中构建的fiber树叫workInProgress fiber，在第一次更新时，所有的更新都发生在workInProgress树，在第一次更新后，workInProgress树上的状态是最新状态，它会替换current树

current:正在视图层渲染的树叫current fiber树

```
currentFiber.alternate = workInProgressFiber
workInProgressFiber.alternate = currentFiber
```

3\. 深度调和子节点，渲染视图

在新建的alternate树上，完成整个子节点的遍历，包括fiber的创建，最后会以workInProgress树最为最新的渲染树，fiberRoot的current指针指向workInProgress使其变成current fiber，完成初始化流程

更新

1. 重新创建workInProgress树，复用当前current树上的alternate，作为新的workInProgress

渲染完成后，workInProgress树又变成current树

双缓冲模式

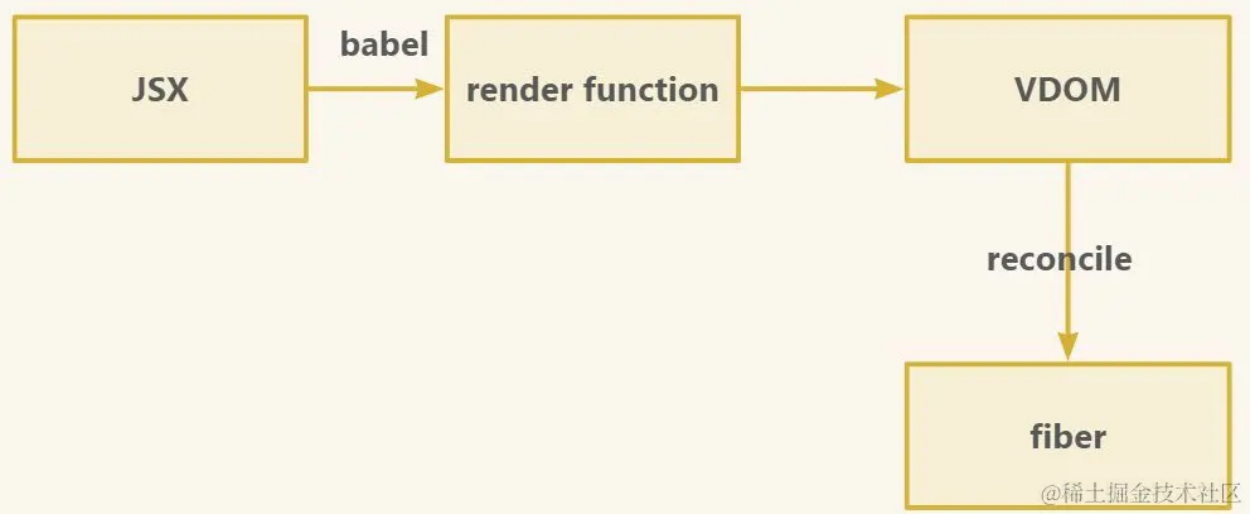
话剧演出中，演员需要切换不同的场景，以一个一小时话剧来说，在舞台中切换场景，时间来不及。一般是准备两个舞台，切换场景从左边舞台到右边舞台演出

在计算机图形领域，通过让图形硬件交替读取两套缓冲数据，可以实现画面的无缝切换，减少视觉的抖动甚至卡顿。

react的current树和workInProgress树使用双缓冲模式，可以减少fiber节点的开销，减少性能损耗

React渲染流程

如图，React用JSX描述页面，JSX经过babel编译为render function，执行后产生VDOM，VDOM不是直接渲染的，会先转换为fiber，再进行渲染。vdom转换为fiber的过程叫reconcile，转换过程会创建DOM，全部转换完成后会一次性commit到DOM，这个过程不是一次性的，而是可打断的，这就是fiber架构的渲染流程



vdom（React Element对象）中只记录了子节点，没有记录兄弟节点，因此渲染不可打断

fiber（fiberNode对象）是一个链表，它记录了父节点、兄弟节点、子节点，因此是可以打断的

原创整理者：**lyllovelemon**