4-react

day01

1. 任务安排

1.1. react介绍
1.2. hello world
1.3. JSX
1.4. 元素渲染
1.5. 生命周期
1.6. 组件
1.7. 事件

2. react简介

https://reactjs.org/

React 是一个声明式，高效且灵活的用于构建用户界面的 JavaScript 库。React起源于 Facebook 的内部项目，用来架设 Instagram 的网站，并于 2013 年 5 月开源。

React 的生态体系比较庞大，它在web端，移动端，桌面端、服务器端，VR领域都有涉及。react可以说是目前为止最热门，生态最完善，应用范围最广的前端框架。react结合它的整个生态，它可以横跨web端，移动端，服务器端，乃至VR领域。可以毫不夸张地说，react已不单纯是一个框架，而是一个行业解决方案。

React特点：

- 声明式设计:React采用声明范式，可以轻松描述应用。
- 高效 :React通过对DOM的模拟，最大限度地减少与DOM的交互。
- 灵活 :React可以与已知的库或框架很好地配合。
- JSX :JSX 是JavaScript语法的扩展。React 开发不一定使用 JSX ，但我们建议使用它。
- 组件:通过 React构建组件，使得代码更加容易得到复用，能够很好的应用在大项目的开发中
- 单向响应的数据流:React 实现了单向响应的数据流，从而减少了重复代码，这也是它为什么比传统数据绑定更简单

3. hello world

我们现在html中引入react，来学习react的核心技术。

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta http-equiv="X-UA-Compatible" content="IE=edge">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>react</title>

  <script src="https://cdn.bootcdn.net/ajax/libs/react/17.0.2/umd/react.proc
  <script src="https://cdn.bootcdn.net/ajax/libs/react-dom/17.0.2/umd/react-
```

```
  <script src="https://cdn.bootcss.com/babel-standalone/6.26.0/babel.min.js'
</head>
<body>
  <div id="app"> </div>
  <script type="text/babel">
    console.log( ReactDOM.render);
    ReactDOM.render(<h1>hello world</h1>, document.getElementById('app'));
  </script>
</body>
</html>
```

4. JSX

JSX，是一个 JavaScript 的语法扩展。建议在 React 中配合使用 JSX，JSX 可以很好地描述 UI 应该呈现出它应有交互的本质形式。JSX 可能会使人联想到模版语言，但它具有 JavaScript 的全部功能。

4.1. jsx中嵌入表达式

```
const name = 'Josh Perez';
const element = <h1>Hello, {name}</h1>;

const obj = {name:"terry",age:12}
const element = <div>{JSON.stringify(obj)}</div>
```

4.2. 动态属性

可以将一个动态值绑定在属性上

```
const element = <img src={user.avatarUrl}></img>;
```

4.3. 子元素

```
const element = (
  <div>
    <h1>Hello!</h1>
    <h2>Good to see you here.</h2>
  </div>
);
```

4.4. JSX表示对象

Babel 会把 JSX 转译成一个名为 `React.createElement()` 函数调用。 以下两个写法完全等价。

```
const element = (
  <h1 className="greeting">
    Hello, world!
  </h1>
);
const element = React.createElement(
  'h1',
  {className: 'greeting'},
  'Hello, world!'
);
```

5. 元素渲染

5.1. react更新

React DOM 会将元素和它的子元素与它们之前的状态进行比较，并只会进行必要的更新来使 DOM 达到预期的状态。

```
<div id="app"></div>
<script type="text/babel">
  function tick(){
    const element = (
      <div>
        <h2>hello tick</h2>
        <div>{new Date().getTime()}</div>
      </div>
    );
    ReactDOM.render(element,document.getElementById('app'));
  }
  setInterval(tick, 1000);
</script>
```

5.2. 条件渲染

```
let flag = Math.random()>0.5;
let element = flag ? <div>大</div>: <div>小</div>
```

5.3. 列表渲染

```
let trs = this.props.list.map((item,index) =>{
 return (
   <tr>
     <td>{index+1}</td>
     <td>{item.title}</td>
     <td>{item.authorId}</td>
     <td>{item.publishTime}</td>
     <td>
       <button onClick={()=>{
         this.props.delArticle(item.id)
       }}>删除</button>
```

```
        <button>修改</button>
      </td>
    </tr>
  )
})
```

6. 生命周期

> 挂载阶段
> 从组件实例被创建到插入 DOM 中的阶段

6.1. constructor(props)

在 React 组件挂载之前，会调用它的构造函数。主要用于初始化state、为实例方法绑定this。如果组件类继承自React.Component，在构造函数中应先调用super(props)，需要注意的是，不用在构造函数中使用this.setState();

```
constructor(props) {
  super(props);
  // Don't call this.setState() here!
  // Don't do this!
  // this.state = { color: props.color };
  this.state = { counter: 0 };
  this.handleClick = this.handleClick.bind(this);
}
```

6.2. static getDerivedStateFromProps(props, state)

响应 Props 变化之后进行更新的方式。这个生命周期的功能实际上就是将传入的props映射到state上面。 该方法是一个静态方法，无法通过this直接访问。这个函数会在每次re-rendering之前被调用。

```
static getDerivedStateFromProps(nextProps, prevState) {
    const {type} = nextProps;
    // 当传入的type发生变化的时候，更新state
    if (type !== prevState.type) {
        return {
            type,
        };
    }
    // 否则，对于state不进行任何操作
    return null;
}
```

6.3. render()

该方法是 class 组件中唯一必须实现的方法。render函数应该是一个纯函数，意味着在不修改state值的时候该函数的返回值是一样的。

6.4. componentDidMount()

会在组件挂载后（插入 DOM 树中）立即调用。依赖于 DOM 节点的初始化应该放在这里。如需通过网络请求获取数据，此处是实例化请求的好地方。这个方法是比较适合添加订阅的地方。如果添加了订阅，请不要忘记在 `componentWillUnmount()` 里取消订阅 。

> 更新阶段

> 当组件的 props 或 state 发生变化时会触发更新。组件就会进入到更新阶段

6.5. static getDerivedStateFromProps(nextProps, nextState)

6.6. shouldComponentUpdate(nextProps, nextState)

根据该方法返回值判断 React 组件的输出是否受当前 state 或 props 更改的影响。默认行为是 state 每次发生变化组件都会重新渲染。大部分情况下，你应该遵循默认行为。

6.7. render()

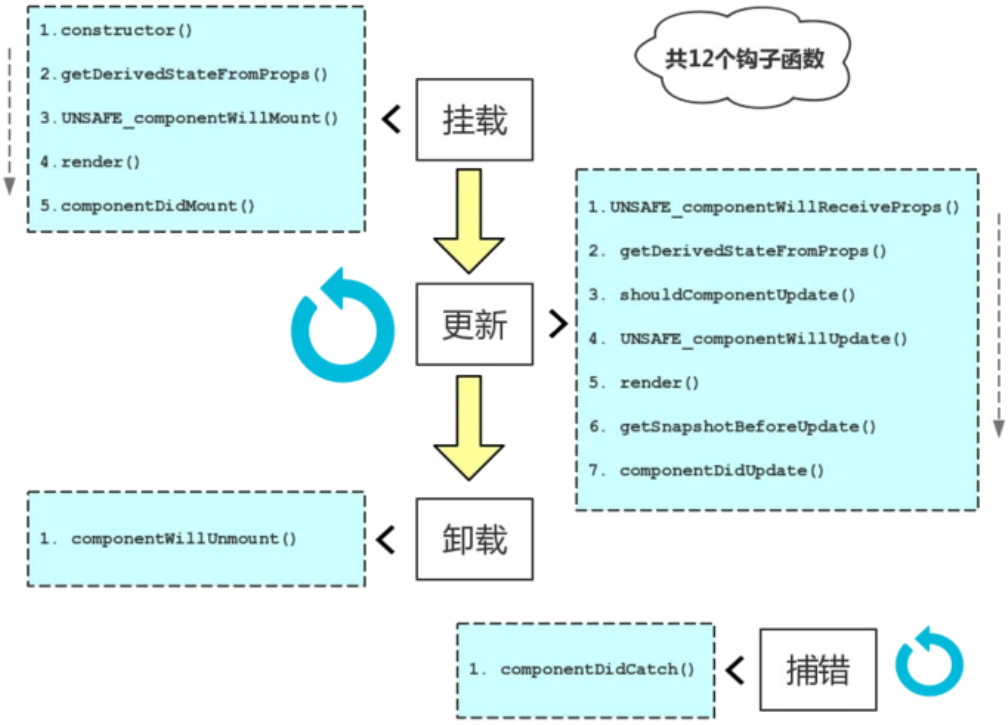
6.8. getSnapshotBeforeUpdate(prevProps, prevState)

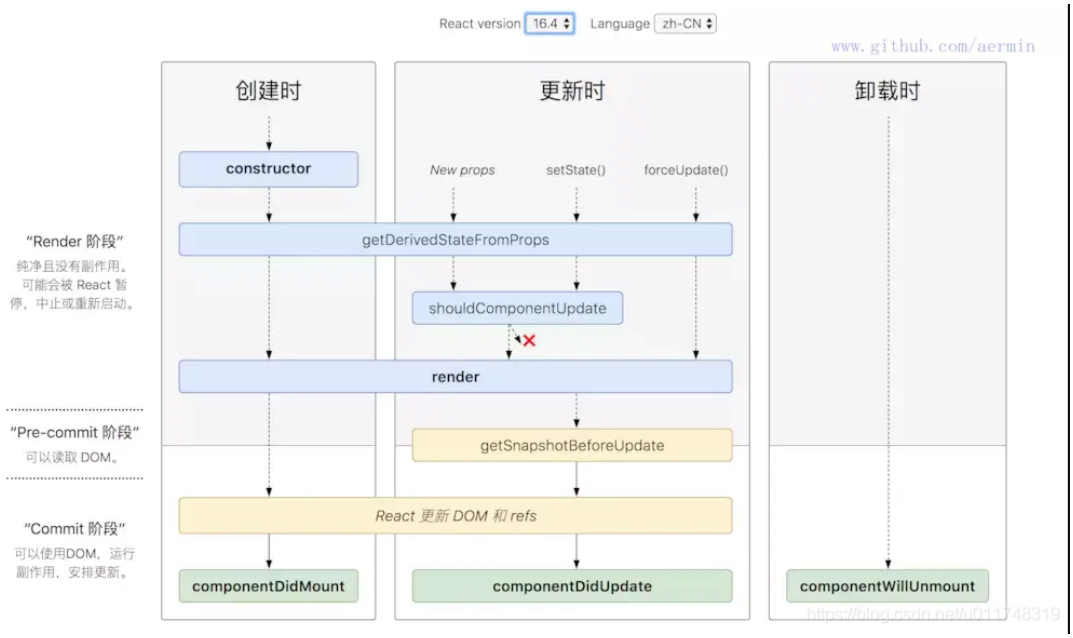
6.9. componentDidUpdate(prevProps, prevState, snapshot)

会在更新后会被立即调用，首次渲染不会执行此方法。当组件更新后，可以在此处对 DOM 进行操作。如果你对更新前后的 props 进行了比较，也可以选择在此处进行网络请求。你也可以在 `componentDidUpdate()` 中**直接调用** `setState()`，但请注意**它必须被包裹在一个条件语句里**，正如上述的例子那样进行处理，否则会导致死循环。它还会导致额外的重新渲染，虽然用户不可见，但会影响组件性能。不要将 props “镜像”给 state，请考虑直接使用 props

> 卸载阶段
> 当组件阶段当组件从 DOM 中移除时

6.10. componentWillUnmount()





7. 组件

7.1. 函数组件

该函数是一个有效的 React 组件，因为它接收唯一带有数据的“props”（代表属性）对象与并返回一个 React 元素。这类组件被称为“函数组件”，因为它本质上就是 JavaScript 函数。相对比类组件，函数组件有以下特点：

- 没有组件实例
- 没有生命周期
- 没有 state 和 setState，只能接收 props
- 函数组件是一个纯函数，执行完即销毁，无法存储 state

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

7.2. 类组件

通过类来定义一个组件。

- 有组件实例
- 有生命周期
- 有 state 和 setState

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

8. 事件

day02

1. 任务安排

1.1. 表单

1.2. 脚手架

1.3. 组件库

1.4. react-router

2. 表单

2.1. 受控组件

在 HTML 中，表单元素（如 `<input>` 、 `<textarea>` 和 `<select>` ）通常自己维护 state，并根据用户输入进行更新。而在 React 中，可变状态（mutable state）通常保存在组件的 state 属性中，并且只能通过使用 `setState()` 来更新。我们可以把两者结合起来，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”

```
class FormContainer extends React.Component {
  constructor(props){
    super(props);
    this.state = {
      form:{
        name:'',
        gender:'男'
      }
    }
    this.handleNameChange = this.handleNameChange.bind(this)
    this.handleGenderChange = this.handleGenderChange.bind(this)
    this.handleProvinceChange = this.handleProvinceChange.bind(this);
  }

  handleNameChange(event){
    this.setState({
      form :{
        ...this.state.form,
        name:event.target.value
      }
    })
  }
  handleGenderChange(event){
    this.setState({
      form :{
        ...this.state.form,
        gender:event.target.value
      }
    })
  }
  handleProvinceChange(event){
    this.setState({
      form :{
        ...this.state.form,
        province:event.target.value
      }
    })
  }

  render() {
```

```
        return (
          <div>
            {JSON.stringify(this.state.form)}
            <form action="">
              <div>
                <label for="">
                  姓名
                  <input type="text" value={this.state.form.name} onChange={
                </label>
              </div>
              <div>
                性别：
                <label for="">男<input type="radio" name="gender" checked={t
                <label for="">女<input type="radio" name="gender" checked={t
              </div>
              <div>
                求职地：
                <select onChange={this.handleProvinceChange}>
                  <option value="上海">上海</option>
                  <option value="苏州">苏州</option>
                  <option value="无锡">无锡</option>
                </select>
              </div>
            </form>
          </div>
        )
      }
    }

    ReactDOM.render(<FormContainer/>,document.getElementById('app'));
```

2.2. 非受控组件

要编写一个非受控组件，而不是为每个状态更新都编写数据处理函数，你可以 使用 ref 来从 DOM 节点中获取表单数据。在非受控组件中，你经常希望 React 能赋予组件一个初始值，但是不去控制后续的更新。 在这种情况下, 你可以指定一个 `defaultValue` 属性，而不是 `value` 。

```
class FormContainer extends React.Component {
     constructor(props){
       super(props);
       this.state = {
         form:{
           name:'',
           gender:'男'
         }
       }
       this.input_name = React.createRef();
       this.input_gender = React.createRef();
       this.select_province = React.createRef();

```

```
        this.handleSubmit = this.handleSubmit.bind(this);
      }

      handleSubmit(event){
        console.log(this.input_name.current.value);
        let form = {
          name:this.input_name.current.value,
          gender:this.input_gender.current.value,
          province:this.select_province.current.value,
        }
        event.preventDefault();
      }

      render() {
        return (
          <div>
            <form action="" onSubmit={this.handleSubmit}>
              <div>
                <label for="">
                  姓名
                  <input type="text" ref={this.input_name} defaultValue="ter
                </label>
              </div>
              <div>
                性别：
                <label for="">男<input type="radio" name="gender" checked va
                <label for="">女<input type="radio" name="gender"  value="女
              </div>
              <div>
                求职地：
                <select ref={this.select_province} defaultValue="苏州">
                  <option value="上海">上海</option>
                  <option value="苏州">苏州</option>
                  <option value="无锡">无锡</option>
                </select>
              </div>
              <div>
                <input type="submit" value="提交"/>
              </div>
            </form>
          </div>
        )
      }
    }

    ReactDOM.render(<FormContainer/>,document.getElementById('app'));
```

3. 脚手架

https://create-react-app.dev/docs/getting-started/

```
# 如果之前安装过create-react-app，要先进行卸载
$ npm uninstall -g create-react-app
$ npx create-react-app my-app
$ cd my-app
$ npm start
```

4. react-router

React Router 是一个基于 React 之上的强大路由库，其中：

react-router是路由库的核心；

react-router-dom是web程序的路由库，在react-router基础上添加了dom操作；

react-router-native是app程序的路由库，在react-router基础上添加了本地操作；

https://reactrouter.com/core/guides/quick-start

4.1. 安装

```
$ cnpm install react-router-dom --save
```

4.2. hello world

```
import { Route ,Switch} from 'react-router'
import { BrowserRouter,Link} from "react-router-dom";
import Home from './pages/Home'
import About from './pages/About';
function App() {
  return (
    <div className="App">
      <BrowserRouter>
        <Link to="/home">home</Link> <br/>
        <Link to="/about">about</Link>
        <hr />
        <Switch>
          <Route path="/home" component={Home}></Route>
          <Route path="/about" component={About}></Route>
        </Switch>
      </BrowserRouter>
    </div>
  );
}

export default App;
```

4.3. 基本组件

- **路由器**

react提供了两种路由器，Browser、Hash，他们主要区别在于存储URL的方式以及与后端交互方式。

`<BrowserRouter>` 类似于history模式，url格式看上去非常和谐，但是服务器需要进行配置，否则会与后端进行交互。

`<HashRouter>` 路径会保存在url后，通过#分割，这种方式不会与后端进行交互

- **路由匹配**

`<Route>` 会将所有匹配的组件进行渲染

`<Switch>` 仅渲染第一个匹配的组件

- **路由导航**

`<Link>` 最基本的路由

`<NavLink>` 带有激活状态的路由

`<Redirect>` 重定向跳转

4.4. 编程式跳转

可以在子组件中访问this.props.hisotry来进行跳转

this.props.history.go()

this.props.history.goBack()

this.props.history.goForward()

this.props.history.push()

```
<button onClick={()=>{
  this.props.history.push('/about')}
}>about</button>
```

4.5. withRouter()

该函数是一个高阶组件，用于为传入的组件的props中注入history、location、match属性。

```
import {Component} from 'react'
import {Redirect,Link,withRouter} from 'react-router-dom'
class BaseNav extends Component{
  constructor(props){
    super(props)
  }
  toArticle(id){
    this.props.history.push('/article/'+id)
  }
  render(){
    return (
      <div>
        <Redirect to="/about"></Redirect>
          <Link to="/home">home</Link> <br/>
          <Link to="/about">about</Link> <br/>
          <Link to="/help">help</Link> <br/>
          <a href="" onClick={(event)=>{
            this.toArticle(9);
            event.preventDefault();
          }}>文章xxx</a> <br/>

      </div>
    )
  }
}
export default withRouter(BaseNav) ;
```

4.6. 参数获取

可以子组件中通过this.props.match.params来获取路由参数，通过this.props.location.query来获取查询字符串参数。

```
componentDidMount(){
   console.log(this.props);
  }
```

day03

1. 任务安排

1.1. Refs

1.2. 高阶组件

1.3. hook

2. Refs

Refs 提供了一种方式，允许我们访问 DOM 节点或在 render 方法中创建的 React 元素。官方建议请勿过渡使用Refs【能不用就别用】

为元素上的ref属性可以绑定一个对象，也可以绑定一个回调函数。当ref属性值为对象的时候，ref会将当前dom赋值给这个对象的current属性；当ref属性会一个函数的时候，ref会将当前dom作为参数传递给这个回调函数。

2.1. 获取原生DOM

```
  class ComponentA extends React.Component {
      constructor(props){
        super(props);
        // this.containerA = {current:null}
        this.containerA = React.createRef(); // {current:null}
        this.containerB = null
      }
      componentDidMount(){
        console.log(this.containerA.current);
        console.log(this.containerB);
      }
      render(){
        return (
          <div>
            <h2>Component A</h2>
            <div ref={this.containerA}>hello component</div>
            <div ref={(e)=>{
              // e就是当前dom对象
              this.containerB = e;
            }}>hello component</div>

          </div>
        )
      }
    }
```

2.2. refs转发

将ref暴露给父组件。可以通过React.forwardRef这个函数来实现，该函数接收一个函数组件作为参数。与其他函数组件不同，该函数组件可以接收2个参数props、ref。这个ref用来将子组件中的dom传递给父组件。

```
class TextInput extends React.Component {
  constructor(props,ref){
    super(props);
    this.ref = ref;
  }
  render(){
    return (
      <div style={ {backgroundColor:'pink'} }>
        <label> 输入框 <input type="text" ref={this.ref}/></label>
      </div>
    )
  }
}
// ref转发。
TextInput = React.forwardRef(TextInput)

// 类组件 表单组件
class FormComponent extends React.Component {
  constructor(props){
    super(props);
    this.submitHandler = this.submitHandler.bind(this);
    this.input_ref = null;
    this.div_ref = null;
  }
  submitHandler(event){
    // 获取表单元素的值？
    console.log('input_ref:',this.input_ref.value);
    event.preventDefault();
  }

  render (){
    return (
      <div>
        <h2>表单</h2>
        <form onSubmit={this.submitHandler}>
          <div> <TextInput ref={(e)=>{
            this.input_ref = e
          }}/> </div>
          <div>
            <input type="submit" value="提交"/>
          </div>
        </form>
      </div>
    )
  }
}
```

3. 高阶组件

HOC（High Order Component） 是 react 中对组件逻辑复用部分进行抽离的高级技术，但HOC并不是一个 React API 。 它只是一种设计模式，类似于装饰器模式。
具体而言，HOC就是一个函数，且该函数接受一个组件作为参数，并返回一个新组件。
从结果论来说，HOC相当于 Vue 中的 mixins(混合)

3.1. 实现机制

- props代理

  将公共代码通过props进行注入

```
function ppHOC(WrappedComponent) {
  PP.staticMethod = WrappedComponent.staticMethod;  // 静态方法
  return class PP extends React.Component {
    render() {
      let props = {
        ...this.props,
        message: "add props"
      };
      return <WrappedComponent {...props} />;
    }
  };
}
```

- 反向继承

  反向继承的方式，除了静态方法之外的生命周期，state，props,render 都可以在 HOC 中得到

```
function ppHOC(WrappedComponent) {
  PP.staticMethod = WrappedComponent.staticMethod;  // 静态方法
  return class PP extends WrappedComponent {
    render() {
      console.log(this.state, "state");
      return super.render();
    }
  };
}
```

3.2. 注意

- 不要在 render 方法中使用 HOC
- 务必复制静态方法
- Refs 不会被传递 ， 可以通过 React.forwardRef 来进行传递

4. Hook

Hook 是 React 16.8 的新增特性。它可以让你在不编写 class 的情况下使用 state 以及其他的 React 特性。Hook 是一些可以让你在函数组件里“钩入” React state 及生命周期等特性的函数。Hook 不能在 class 组件中使用 —— 这使得你不使用 class 也能使用 React。可以在新组件中慢慢使用Hook。

在函数组件中，我们无法使用this，因此也就无法使用this.state来保存局部状态，Hook出现可以让我们函数组件中定义状态。

```
const useState = React.useState;
function Counter (){
```

```
  const [count,setCount] = useState(0);
  return (
    <div>
      你点击了{count} 次
      <button onClick={()=>{setCount(count+1)}}>点击</button>
    </div>
  )
}
ReactDOM.render(<Counter/>,document.getElementById('app'));
```

等价

```
class Counter extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      count: 0
    };
  }
  render() {
    return (
      <div>
        <p>You clicked {this.state.count} times</p>
        <button onClick={() => this.setState({ count: this.state.count + 1 }
          Click me
        </button>
      </div>
    );
  }
}
```

4.1. State Hook

useState用于声明一个state变量，该函数只有一个参数，表示声明的变量的初始值。每次调用useState可以创建一个state变量，调用多次可以创建多个state变量。改方法的返回值为state变量及更新该state变量的函数，可以通过数组解构来获取。

```
// 声明一个叫 “count” 的 state 变量
const [count, setCount] = useState(0);

```

4.2. Effect Hook

为什么使用EffectHook？在函数组件主体内（这里指在 React 渲染阶段）改变 DOM、添加订阅、设置定时器、记录日志以及执行其他包含副作用的操作都是不被允许的，因为这可能会产生莫名其妙的 bug 并破坏 UI 的一致性。

- **useEffect**

Effect Hook 可以让你在函数组件中执行副作用操作。useEffect 就是一个 Effect Hook，给函数组件增加了操作副作用的能力。它跟 class 组件中的 componentDidMount、componentDidUpdate 和 componentWillUnmount 具有相同的用途，只不过被合并成了一个 API。

useEffect这个函数可以接收两个参数，第一个为函数；第二个为数组。对于第二个参数，如果你传入了一个空数组（`[]`），effect 内部的 props 和 state 就会一直持有其初始值。可

以向数组中添加需要监听的state，当该state发生变化会引起useEffect重新执行。useEffect 在渲染时是异步执行，并且要等到浏览器将所有变化渲染到屏幕后才会被执行。

- **useLayoutEffect**

其函数签名与 `useEffect` 相同，但它会在所有的 DOM 变更之后同步调用 effect。可以使用它来读取 DOM 布局并同步触发重渲染。在浏览器执行绘制之前，`useLayoutEffect` 内部的更新计划将被同步刷新。尽可能使用标准的 `useEffect` 以避免阻塞视觉更新。

```
import React, { useState, useEffect } from 'react';
function Example() {
  const [count, setCount] = useState(0);
  useEffect(() => {    document.title = `You clicked ${count} times`;  });
  return (
    <div>
      <p>You clicked {count} times</p>
      <button onClick={() => setCount(count + 1)}>
        Click me
      </button>
    </div>
  );
}
```

- **无需清除的 effect**

有时候，我们只想**在 React 更新 DOM 之后运行一些额外的代码。**比如发送网络请求，手动变更 DOM，记录日志，这些都是常见的无需清除的操作。因为我们在执行完这些操作之后，就可以忽略他们了。默认情况下，effect在第一次渲染之后*和*每次更新之后都会执行。

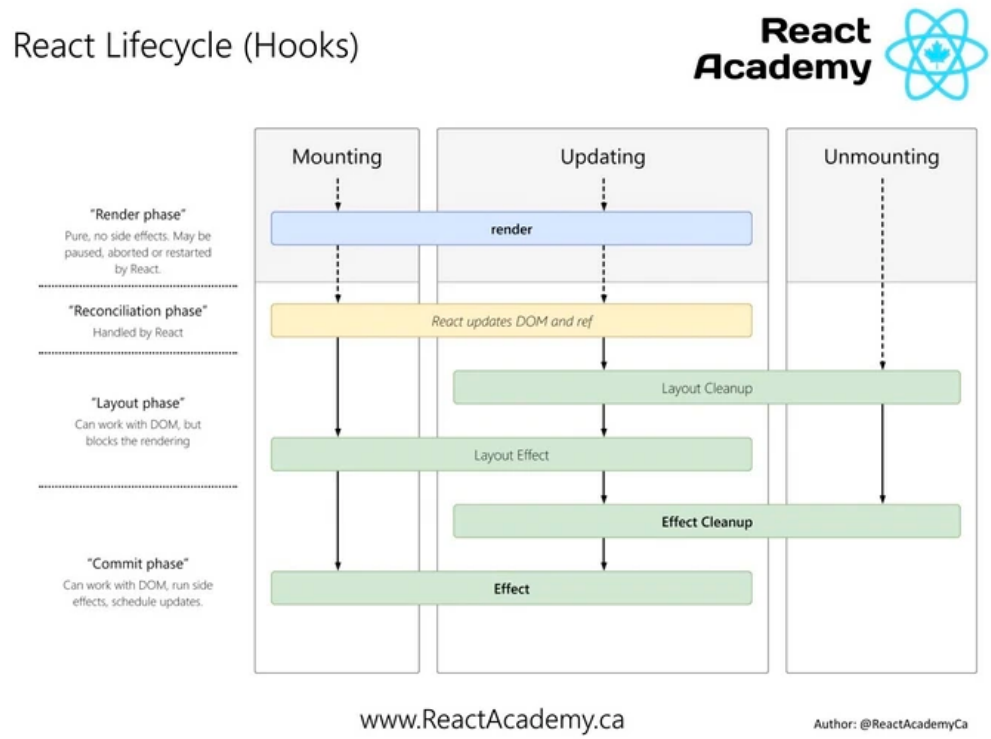
```
useEffect(() => {
  document.title = `You clicked ${count} times`;
});
```

- **需要清除的 effect**

有些副作用是需要清除的。例如**订阅外部数据源**、渲染图表。这种情况下，清除工作是非常重要的，可以防止引起内存泄露！如果你的 effect 返回一个函数，React 将会在执行清除操作时调用它。

```
useEffect(() => {
  function handleStatusChange(status) {

  }
  return function cleanup() {
  };
});
```



4.3. 其他Hook

- useMemo

与useEffect有点类似，只有useMemo中的第二个参数[count]发生变化的时候，才会去从新执行expensive方法

```
function Counter(props){
  const [count,setCount] = useState(100)
  // 如果不使用useMemo,每次num更新该函数也会执行。
  let total = useMemo(()=>{
    console.log('computed...',Math.random());
    let result = 0;
    for(let i=1;i<=100;i++){
      result += i;
    }
    return result;
  },[count]);

  const [num,setNum] = useState(0)

  return (
    <div>
      <h2>counter</h2>
      <div>total : {total}</div>
      <div>{num}</div>
      <div><button onClick={()=>{setNum(num+1)}}>change</button></div>
    </div>
  )
```

```
}
```

- useCallback

4.4. 自定义Hook

```
function useClock(){
  let [date,setDate] = useState(new Date());
  useEffect(()=>{
    const timmer = setInterval(() => {
      setDate(new Date)
    }, 1000);
    return ()=>{
      clearInterval(timmer)
    }
  },[])
  return date;
}

<div>{useClock().toLocaleTimeString()}</div>
```

day04

1. 任务安排

2. antd安装

3. 表格组件

3.1. 基本应用

表格的应用非常简单，可以接收至少3个参数：dataSource数据源，columns列定义，rowKey；常见的Table属性如下：【更详细请查看官方文档】

dataSource	数据源，为一个数组，数组中可以存放多个对象
columns	列定义 • title 表头文本 • align 对齐方式 • dataIndex 列数据在数据项中对应的路径，支持通过数组查询嵌套路径，类似于el中的prop • key 唯一标识，React 需要的 key • render 自定义渲染函数，function(text, record, index) {}。参数分别为：当前列、当前行、索引 • width 列宽度
bordered	是否展示外边框和列边框
rowKey	表格行 key 的取值，可以是字符串或一个函数
loading	页面是否加载中
size	表格大小 default/middle/small
onChange	分页、排序、筛选变化时触发
pagination	

```
render(){
  const columns = [
    { title: '标题', dataIndex: 'title' ,key:'title'},
    { title: '发布时间', dataIndex: 'publishTime',key:'publishTime'},
    { title: '状态', dataIndex: 'status' , key:'status'},
    { title: '作者', dataIndex: 'authorId', key:'authorId' },
    {
      title: '操作',
      dataIndex:'id',
      key: 'action',
      align:'center',
      render: (text, record) => (
        <Space size="middle" align="center">
          <a>Invite</a>
          <a>Delete</a>
        </Space>
      ),
    },
  ]

  return (
    <div>
      <h2>文章管理</h2>
      <Button type="primary">发布</Button>
      <Table
        rowKey={record=>record.id}
        size="small"
        dataSource={this.state.articles.list}
        columns={columns} />;
    </div>
  )
```

```
  }

```

注意：我们在componentDidMount中进行ajax，当组件切换的时候可能出现ajax响应还未回来，待响应回来后还需setState，这便引起了异常

```
componentWillUnmount(){
  this.setState = ()=>false;
}
```

3.2. 分页

表格分页也很简单，我们需要为Table添加pagination属性，该属性值为一个对象，对象中的参数可以参照：https://ant.design/components/pagination-cn/

current	当前页
pageSize	每页条数
total	总数
onChange	页码或 pageSize 改变的回调，参数是改变后的页码及每页条数

```
<Table
  pagination={{
    current:this.state.articles.page,
    pageSize:this.state.articles.pageSize,
    total:this.state.articles.total,
    onChange:this.pageChangeHandler
  }}
  rowKey={record=>record.id}
  size="small"
  dataSource={this.state.articles.list}
  columns={columns}
/>;
```

4. 表单组件

4.1. 基本应用

表单用于收集用户信息，我们主要使用表单的默认值、收集表单数据功能。我们常把表单组件封装到函数组件中。这里主要用到Form组件，该组件常见属性如下：

name	表单名称
form	经 Form.useForm() 创建的 form 控制实例，不提供时会自动创建
colon	label后是否添加冒号，默认为true
labelAlign	label 标签的文本对齐方式 left/right，默认为right
layout	表单布局，horizontal（默认值） | vertical | inline
size	表单大小，small | middle | large
initialValues	表单默认值，只有初始化以及重置时生效
onFinish	提交表单且数据验证成功后回调事件
onFinishFailed	提交表单且数据验证失败后回调事件

validateMessages	验证提示模板
	const validateMessages = {
	required: "'${name}' 是必选字段",
	// ...
	};

表单元素的子元素通常是Form.Item，用于数据双向绑定、校验、布局等

label	标签的文本
name	字段名，支持数组
rules	校验规则，设置字段的校验逻辑
wrapperCol	需要为输入控件设置布局样式时，你可以通过 Form 的 wrapperCol 进行统一设置，

```

function ArticleForm(){
  const onFinish = (values) => {
    console.log('Success:', values);
  };

  const onFinishFailed = (errorInfo) => {
    console.log('Failed:', errorInfo);
  };

  return (
    <Form
      name="basic"
      initialValues={{
        title:'默认标题',
        content:'默认内容'
      }}
      onFinish={onFinish}
      onFinishFailed={onFinishFailed}
    >
      <Form.Item
        label="标题"
```

```
        name="title"
        rules={[
          {
            required: true,
            message: 'Please input content title!',
          },
        ]}
      >
        <Input />
      </Form.Item>

      <Form.Item
        label="内容"
        name="content"
        rules={[
          {
            required: true,
            message: 'Please input article content!',
          },
        ]}
      >
        <Input.TextArea />
      </Form.Item>

      <Form.Item>
        <Button type="primary" htmlType="submit">
          Submit
        </Button>
      </Form.Item>
    </Form>
  )
}

export default ArticleForm;
```

4.2. 模态框表单

antd推荐使用 `Form.useForm` 创建表单数据域进行控制 。将Form嵌套在Modal中，Modal常见的属性如下：

visible	对话框是否可见
title	标题
okText	确认按钮文字
cancelText	取消按钮文字
onCancel	点击遮罩层或右上角叉或取消按钮的回调
onOk	点击确定回调
width	宽度

```
const ArticleFormModal = ({ visible, onCreate, onCancel }) => {
```

```
  const [form] = Form.useForm();
  return (
    <Modal
      visible={visible}
      title="Create a new collection"
      okText="保存"
      cancelText="取消"
      onCancel={onCancel}
      onOk={() => {
        form
          .validateFields()
          .then((values) => {
            form.resetFields();
            onCreate(values);
          })
          .catch((info) => {
            console.log('Validate Failed:', info);
          });
      }}
    >
      <Form
        form={form}
        layout="vertical"
        name="form_in_modal"
        initialValues={{
          title:'默认标题',
        content:'默认内容'
        }}
      >
        <Form.Item
          label="标题"
          name="title"
          rules={[
            {
              required: true,
              message: 'Please input content title!',
            },
          ]}
        >
          <Input />
        </Form.Item>

        <Form.Item
          label="内容"
          name="content"
          rules={[
            {
              required: true,
              message: 'Please input article content!',
```

```
            },
          ]}
        >
          <Input.TextArea />
        </Form.Item>
      </Form>
    </Modal>
  );
};
```

5. 提示框

5.1. 全局提示

message具有success、error、warning、info、loading等方法，分别表示成功提示、错误提示、警告提示、普通提示、加载提示

```
import { message} from 'antd';
const success = () => {
  message.success('This is a success message');
};
```

5.2. 通知提示框

与message类似，notification也具有success、error、warning、info等方法

```
 import { notification } from 'antd';
 notification.open({
    message: 'Notification Title',
    description:
      'This is the content of the notification. This is the content of the r
    onClick: () => {
      console.log('Notification Clicked!');
    },
  });

```

5.3. 确认对话框

```
import { Modal} from 'antd';
const { confirm } = Modal;
function showConfirm() {
  confirm({
    title: 'Do you Want to delete these items?',
    icon: <ExclamationCircleOutlined />,
    content: 'Some descriptions',
    onOk() {
      console.log('OK');
    },
    onCancel() {
      console.log('Cancel');
    },
  });
}
```

6. 综合案例应用

行学天下门户系统开发

day05

1. 任务安排

1.1. redux介绍

1.2. redux核心概念

1.3. redux hello world

2. redux介绍

http://cn.redux.js.org/introduction/getting-started

https://redux.js.org/

https://react-redux.js.org/

Redux 是一个有用的架构，但不是非用不可。事实上，大多数情况，你可以不用它，只用 React 就够了。

需要用到redux的情况：

- 用户的使用方式复杂
- 不同身份的用户有不同的使用方式（比如普通用户和管理员）
- 多个用户之间可以协作
- 与服务器大量交互，或者使用了WebSocket
- View要从多个来源获取数据
- 某个组件的状态，需要共享
- 某个状态需要在任何地方都可以拿到
- 一个组件需要改变全局状态
- 一个组件需要改变另一个组件的状态

3. 核心概念

3.1. store

Store 就是保存数据的地方，你可以把它看成一个容器。整个应用只能有一个 Store。Redux 提供createStore这个函数，用来生成 Store。

3.2. state

托管给redux管理的状态

```
let state = {
  todos:[],
  params:{}
}
```

3.3. action

Action 描述当前发生的事情。改变 State 的唯一办法，就是使用 Action。它会运送数据到 Store。Action 本质上是 JavaScript 普通对象。我们约定，action 内必须使用一个字符串类型的 `type` 字段来表示将要执行的动作。 **Action 创建函数** 就是生成 action 的方法。

```
{ type: 'ADD_TODO', text: '去游泳馆' }
{ type: 'TOGGLE_TODO', index: 1 }
{ type: 'SET_VISIBILITY_FILTER', filter: 'completed' }
```

3.4. reducer

Reducers 指定了应用状态的变化如何响应 actions 并发送到 store 的。reducer 只是一个接收 state 和 action，并返回新的 state 的纯函数。 对于大的应用来说，不大可能仅仅只写一个这样的函数，所以我们编写很多小函数来分别管理 state 的一部分：

```
// reducer
function todos(state = [],action){
  switch(action.type){
    case "ADD_TODO":
      return [
        ...state,
        action.payload
      ]
    case "REMOVE_TODO":
      return state.filter(item => item.id !== action.payload);
    default:
      // 默认务必返回state
      return state;
  }
}
```

4. React中应用Redux

需要先创建一个store，然后将store注入给react组件，在组件中通过props来访问state以及通过props来dispatch action。

4.1. 配置store

声明action以及reducer

```
//1. action
let addTodo = (todo)=>{
  return {
    type:"ADD_TODO",
    payload:todo
  }
}
let removeTodo = (id)=>{
  return {
    type:"REMOVE_TODO",
    payload:id
  }
}
let setParams = (params)=>{
  return {
    type:'SET_PARAMS',
    payload :params
  }
}
//2. reducer : 这里的state就是状态
function todos(state = [],action){
  switch(action.type){
    case "ADD_TODO":
      return [
```

```
        ...state,
        action.payload
      ]
    case "REMOVE_TODO":
      return state.filter(item => item.id !== action.payload);
    default:
      return state;
  }
}
function params(state = {},action){
  switch(action.type){
    case "SET_PARAMS":
      return action.payload;
    default :
      return state;
  }
}
```

4.2. 创建store实例

先通过combineReducers组合多个Reducer，然后通过createStore来创建状态机。

```
/ 生成store
let store = Redux.createStore(
  Redux.combineReducers({ params, todos }))
```

4.3. 注入到根组件中

通过Provider将状态机绑定到根组件上，这样子组件中都可以应用状态机

```
let Provider = ReactRedux.Provider;

ReactDOM.render(
  <Provider store={store}>
    <TodoList/>
  </Provider>
  ,
  document.getElementById('app')
);
```

4.4. 子组件映射state及action

connect函数可以接收两个函数：mapStateToProps，mapDispatchToProps，分别用于映射state以及action，这两个函数均需要返回一个对象

```
TodoList = ReactRedux.connect(
  (state)=>{
    return {
      todos:state.todos
    }
  },
  (dispatch)=>{
    return {
      addTodo:todo =>{
```

```
        dispatch(addTodo(todo))
      }
    }
  }
)(TodoList);
```

5. Redux API

5.1. 顶级 API

- createStore(reducer, [preloadedState], [enhancer])
- combineReducers(reducers)
- applyMiddleware(...middlewares)

5.2. store API

- getState()
- dispatch(action)
- subscribe(listener)
- replaceReducer(nextReducer)

5.3. React-Redux API

- connect(mapStateToProps?, mapDispatchToProps?, mergeProps?, options?)

6. 中间件

6.1. 中间件应用

redux中间件用于拓展redux功能。要想使用中间件也非常简单，1：创建中间件，2：应用中间件

6.2. redux-saga

在html中引用redux-saga

```
<script src="https://cdn.bootcdn.net/ajax/libs/redux-saga/1.1.3/redux-saga.umd.js"></script>

<script  data-plugins="transform-es2015-modules-umd"  type="text/babel">
    import 'regenerator-runtime/runtime'
</script>
```

1. **定义副作用，并且映射为action**

```
let {put,takeEvery} = ReduxSaga.effects
function* fetchUser(action) {
  let url = 'http://121.199.29.84:8001/index/carousel/findAll'
  let resp =yield axios.get(url);
  yield put(setTodo(resp.data.data))
}
let mySaga = function* mySaga() {
  yield takeEvery("USER_FETCH_REQUESTED", fetchUser);
}
```

2. **应用中间件**

```
const sagaMiddleware = ReduxSaga.default();
```

```
let store = Redux.createStore(
  Redux.combineReducers({ params, todos }),
  Redux.applyMiddleware(sagaMiddleware)
)
sagaMiddleware.run(mySaga)
```

3. **调用异步操作**

```
dispatch({type:'USER_FETCH_REQUESTED'})
```

6.3. redux-saga API