React 全家桶（技术栈）

1. 第一章：React 入门

1.1. React 简介

1.1.1. 官网

1. 英文官网：https://reactjs.org/
2. 中文官网: https://react.docschina.org/

1.1.2. 介绍描述

1. 用于动态构建用户界面的 JavaScript 库（只关注于视图）
2. 是一个将数据渲染为HTML视图的开源 JavaScript 库
3. 由 Facebook 开发，且开源

1.1.3. 原生 JavaScript 的缺点

1. 原生 JavaScript 操作 DOM 繁琐、效率低（DOM-API 操作 DOM）。
2. 使用 JavaScript 直接操作 DOM，浏览器会进行大量的重绘重排。
3. 原生 JavaScript 没有组件化编码方案，代码复用率低。

1.1.4. React 的特点

1. 声明式编码
2. 组件化编码
3. React Native 编写原生应用
4. 高效（优秀的 Diffing 算法）

> 1. 采用组件化编码、声明式编码（以前的编码方式是命令式编码），提高开发效率及组件复用率。
> 2. 在React Native中可以使用 React 语法进行移动端开发
> 3. 使用虚拟DOM（没放在页面上，放在代码运行时的电脑内存里） + 优秀的 Diffing 算法，尽量减少与真实 DOM 的交互

1.1.5. React 高效的原因

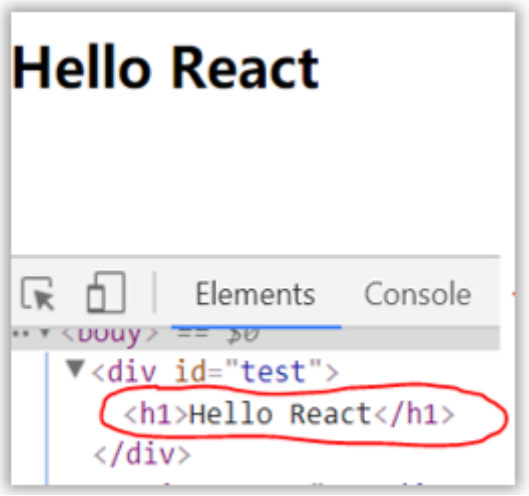
1. 使用虚拟(virtual)DOM，不总是直接操作页面真实DOM。
2. DOM Diffing 算法，最小化页面重绘。

1.1.6. React 学习前置知识

- 判断 this 的指向
- class（类）
- ES6 语法规范
- npm 包管理器
- 原型、原型链
- 数组常用方法
- 模块化

1.2. React 的基本使用

1.2.1. 效果



1.2.2. 相关 js 库

1. react.js：React 核心库。
2. react-dom.js：提供操作 DOM 的 react 扩展库。
3. babel.min.js：解析 JSX 语法代码转为 JS 代码的库。

1.2.3. React 基本使用代码示例

```
<!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>hello_react</title>
  </head>
  <body>
    <!-- 准备好一个“容器” -->
    <div id="test"></div>

    <!-- 引入react核心库 -->
    <script type="text/javascript" src="../js/react.development.js">
</script>
    <!-- 引入react-dom，用于支持react操作DOM -->
    <script type="text/javascript" src="../js/react-dom.development.js">
</script>
    <!-- 引入babel，用于将jsx转为js -->
    <script type="text/javascript" src="../js/babel.min.js"></script>


    <script type="text/babel">  // 此处一定要写babel
      // 1.创建虚拟DOM
      const VDOM = <h1>Hello, React</h1> // 此处一定不要写引号，因为不是字符串
      // 2.渲染虚拟DOM到页面
      ReactDOM.render(VDOM, document.getElementById('test'))
    </script>
  </body>
</html>
```

1.2.4. 创建虚拟 DOM 的两种方式

1. 纯 JS 方式（一般不用）

```
<!-- 准备好一个“容器” -->
  <div id="test"></div>

  <!-- 引入react核心库 -->
  <script type="text/javascript" src="../js/react.development.js">
</script>
  <!-- 引入react-dom，用于支持react操作DOM -->
  <script type="text/javascript" src="../js/react-dom.development.js">
</script>

  <script type="text/javascript"> // 此处一定要写babel
    // 1.创建虚拟DOM
    // 语法：const VDOM = React.createElement(标签名，标签属性，标签内容)
    const VDOM = React.createElement('h1', {id: 'title'}, 'Hello React')
    // 2.渲染虚拟DOM到页面
    ReactDOM.render(VDOM, document.getElementById('test'))
  </script>
```

2. JSX 方式

```
<!-- 准备好一个“容器” -->
  <div id="test"></div>

  <!-- 引入react核心库 -->
  <script type="text/javascript" src="../js/react.development.js">
</script>
  <!-- 引入react-dom，用于支持react操作DOM -->
  <script type="text/javascript" src="../js/react-dom.development.js">
</script>
  <!-- 引入babel，用于将jsx转为js -->
  <script type="text/javascript" src="../js/babel.min.js"></script>

  <script type="text/babel">  // 此处一定要写babel
    // 1.创建虚拟DOM
    const VDOM = ( /* 此处一定不要写引号，因为不是字符串 */
      <h1 id="title">
        <span>Hello React</span>
      </h1>
    )
    // 2.渲染虚拟DOM到页面
    ReactDOM.render(VDOM, document.getElementById('test'))
  </script>
```

1.2.5. 虚拟 DOM 与真实 DOM

关于虚拟DOM：

- 本质是object类型的对象(一般对象)
- 虚拟DOM比较轻，真实DOM比较重，因为虚拟DOM是React内部在用，无需真实DOM上那么多的属性。
- 虚拟DOM最终会被React转化为真实DOM，呈现在页面上。

1. React 提供了一些 API 来创建一种“特别”的一般 js 对象

```
const VDOM = React.createElement('h1', {id: 'title'}, 'Hello React')
```

上面创建的就是一个简单的虚拟 DOM 对象

2. 我们编码时基本只需要操作 react 的虚拟 DOM 相关数据，react 会转换为真是 DOM 变化而更新界面。

1.3. React JSX

1.3.1. 效果

1.3.2. JSX

1. 全称：JavaScript XML
2. react 定义了一种 类似于 XML 的 JS 扩展语法：JS + XML，本质是 React.createElement(component, props, ...children) 方法的语法糖
3. 作用：用来简化创建虚拟DOM
   - 写法：`var els = <h1>Hello JSX</h1>`
   - 注意1：他不是字符串，也不是 HTML/XML 标签
   - 注意2：它最终产生一个 JS 对象
4. 标签名任意：HTML标签或其它标签
5. 便签名属性任意：HTML　标签属性或其它
6. 基本语法规则
   - 定义虚拟DOM时，不要写引号。
   - 标签重混入 JS 表达式时要用 {} 包裹起来。
   - 样式的类名指定不要用 class，要用 className。
   - 内联样式，要用 `style={{key:value}}` 的形式去写。
   - 只有一个根标签。
   - 标签必须闭合
   - 标签首字母
     - 若小写字母开头，则将该标签转为html中同名元素，若html中无该标签对应的同名元素，则报错。
     - 若大写字母开头，react 就去渲染对应的组件，若组件没有定义，则报错 。
   - 遇到 < 开头的代码，以标签的语法解析：html 同名标签转换为 html 同名元素，其它标签需要特别解析

- 遇到以 { 开头的代码，以 JS 语法解析：标签中的 js 表达式必须用 {} 包含

7. babel.js 的作用
   - 浏览器不能直接解析 JSX 代码，需要 babel 转译为 纯 JS 的代码才能运行
   - 只要用了 JSX，都要加上 `type = "text/babel"`，声明需要 babel 来处理

1.3.3. 渲染虚拟 DOM(元素)

1. 语法：

```
// 1. 创建虚拟DOM
const VDOM = <h1>hello react</h1>
// 2.渲染虚拟DOM到页面
ReactDOM.render(VDOM, document.getElementById('root'))
```

2. 作用：将虚拟 DOM 元素渲染到页面中的真实容器 DOM 中显示
3. 参数说明：
   - virtualDOM：纯 js 或 jsx 创建的虚拟 dom 对象
   - containerDOM：用来包含虚拟 DOM 元素的真实 dom 元素对象（一般是一个div）

1.3.4. JSX 练习

代码实现：

```
<!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>
  </head>
  <body>
    <!-- 准备好一个“容器” -->
    <div id="root"></div>

    <!-- 引入react核心库 -->
    <script type="text/javascript" src="../js/react.development.js">
</script>
    <!-- 引入react-dom，用于支持react操作DOM -->
    <script type="text/javascript" src="../js/react-dom.development.js">
</script>
    <!-- 引入babel，用于将jsx转为js -->
    <script type="text/javascript" src="../js/babel.min.js"></script>

```

```
    <script type="text/babel">
      /*
        一定要注意区分：【js语句(代码) 与 【js表达式】
          1.表达式：一个表达式会产生一个值，可以放在任何一个需要值的地方
            下面这些都是表达式：
              (1).a
              (2).a+b
              (3).demo(1)
              (4).arr.map()
              (5).function test()
          2.语句(代码)：
            下面这些都是语句(代码)：
              (1).if(){}
              (2).for(){}
              (3).switch(){case:xxx}
       */

      // 模拟一些数据
      const data = ['Angular', 'React', 'Vue']
      // 1.创建虚拟DOM
      const VDOM = (
        <div>
          <h1>前端js框架列表</h1>
          <ul>
            {
              data.map((item, index) => <li key={index}>{item}</li>)
            }
          </ul>
        </div>
      )
      // 2. 渲染虚拟DOM到页面
      ReactDOM.render(VDOM, document.getElementById('root'))
    </script>
  </body>
</html>
```

1.4. 模块与组件、模块化与组件化的理解

1.4.1. 模块

1. 理解：向外提供特定功能的 js 程序，一般一个 js 文件就是一个模块。
2. 为什么要拆成模块：随着业务逻辑的增加，代码越来越多且复杂。
3. 作用：复用 js，简化 js 的编写，提高 js 运行效率。

1.4.2. 组件

1. 理解：用来实现局部功能效果的代码和资源的集合(html/css/js/image 等等)。
2. 为什么要用组件：一个界面功能更复杂
3. 作用：复用编码，简化项目代码，提高运行效率

1.4.3. 模块化

当应用的 js 都是以模块来编写的，这个应用就是一个模块化的应用。

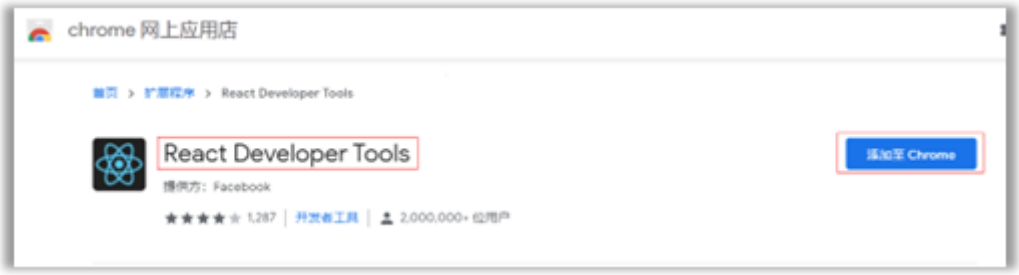
1.4.4. 组件化

当应用是以多组件的方式实现，这个应用就是一个组件化应用

2. 第二章：React 面向组件编程

2.1. 基本理解和使用

2.1.1. 使用 React 开发者工具调试



2.1.2. 函数式组件

函数组件代码示例：

```
<script type="text/babel">
  // 创建函数式组件
  function MyComponent() {
    console.log(this) // 此处的this是undefined，因为babel编译后开启了严格模式
    return <h2>我是用函数定义的组件(适用于【简单组件】的定义)</h2>
  }
  // 渲染组件到页面
  ReactDOM.render(<MyComponent />, document.getElementById('root'))
  /*
    执行了ReactDOM.render(<MyComponent />, document.getElementById('root'))之后，发生了什么？
      1.React解析组件标签，找到了Mycomponent组件。
      2.发现组件时使用函数定义的，随后调用该函数，将返回的虚拟DOM转为真实DOM，随后呈现在页面中。
  */
</script>
```

2.1.3. 类式组件

类相关知识复习：

1. 类中的构造器不是必须写的，要对实例进行一些初始化的操作，如添加指定属性是才写。
2. 如果A类继承了B类，且A类中写了构造器，那么A类构造器中super是必须要调用的。
3. 类中所定义的方法，都是放在了类的原型对象上，供实例去使用

类式组件代码示例：

```
<script type="text/babel">
  // 1.创建类式组件
  class MyComponent extends React.Component {
    render() {
      // render是放在哪里的？—— 类（MyComponent）的原型对象上，供实例使用。
      // render中的this是谁？—— MyComponent的实例对象 <=> MyComponent组件实例对象
      return <h2>我是用类定义的组件(适用于【复杂组件】的定义)</h2>
    }
  }
  // 2.渲染组件到页面
  ReactDOM.render(<MyComponent/>, document.getElementById('root'))
  /*
    执行了ReactDOM.render(<MyComponent/>, document.getElementById('root'))之后，发生了什么？
      1.React解析组件标签，找到了Mycomponent组件。
      2.发现组件时使用类定义的，随后new出来该类的实例，并通过该实例调用到原型上的render方法。
      3.将render返回的虚拟DOM转为真实DOM，随后呈现在页面中。
  */
```

2.1.4. 注意：

1. 组件名必须首字母大写
2. 虚拟 DOM 元素只能有一个根元素
3. 虚拟 DOM 元素必须有结束标签

2.1.5. 渲染类组件标签的基本流程

1. React 内部会创建组件实例对象
2. 调用 render() 得到虚拟 DOM，并解析为真实 DOM
3. 插入到指定的页面元素内部

2.2. 组件实例的三大核心属性 1：state

组件的状态驱动着页面展示

状态在哪里，操作状态的方法就在哪里

简单组件和复杂组件定义：简单组件没有状态，复杂组件有状态

2.2.1. 效果

需求：定义一个展示天气信息的组件

1. 默认展示天气凉爽或炎热
2. 点击文字切换天气

代码实现：

```
<!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>
  </head>
```

```
  <body>
    <!-- 准备好一个“容器” -->
    <div id="root"></div>

    <!-- 引入react核心库 -->
    <script type="text/javascript" src="../js/react.development.js">
</script>
    <!-- 引入react-dom，用于支持react操作DOM -->
    <script type="text/javascript" src="../js/react-dom.development.js">
</script>
    <!-- 引入babel，用于将jsx转为js -->
    <script type="text/javascript" src="../js/babel.min.js"></script>

    <script type="text/babel">
      // 创建组件
      class Weather extends React.Component {
        constructor(props) {
          console.log('constructor')
          // 构造器调用几次？ —— 1次
          super(props)
          // 初始化状态
          this.state = { isHot: false, wind: '微风' }
          // 解决changeWeather中this指向问题
          // 后面表示顺着Weather的原型链找到的changeWeather再使用bind改变this指向，
之后赋值挂在自身身上一个changeWeather
          this.changeWeather = this.changeWeather.bind(this)
        }

        //render调用几次？ —— 1+n 次，1是初始化的那次，n是状态更新的次数
        render() {
          console.log('render')
          // 读取状态
          const { isHot, wind } = this.state
          return (
            <h1 onClick={this.changeWeather}>
              今天天气很{isHot ? '炎热' : '凉爽'}, {wind}
            </h1>
          )
        }

        // changeWeather调用几次？ —— 点几次调几次
        changeWeather() {
          // changeWeather 放在哪里？ —— Weather 的原型对象上，供实例使用
          // 由于changeWeather是作为onClick的回调，所以不是通过实例调用的，是直接调用
          // 类中的方法默认开启了局部的严格模式，所以changeWeather中的this为
undefined

          console.log('changeWeather')
          // 获取原来的isHot值
          const isHot = this.state.isHot
          // 严重注意：状态必须通过setState进行更新，且是合并
          this.setState({ isHot: !isHot })

          // 严重注意：状态(state)不可直接更改，下面这行就是直接更改！！！
          // this.state.isHot = !isHot // 这是错误的写法
        }
      }
      // 渲染组件到页面
```

```
      ReactDOM.render(<Weather />, document.getElementById('root'))
    </script>
  </body>
</html>
```

state的简写方式：

```
<script type="text/babel">
  class Weather extends React.Component {
    // 初始化状态
    state = { isHot: false, wind: '微风' }

    render() {
      const { isHot, wind } = this.state
      return (
        <h1 onClick={this.changeWeather}>
          今天天气很{isHot ? '炎热' : '凉爽'}, {wind}
        </h1>
      )
    }

    // 自定义方法----要用赋值语句的形式+箭头函数
    changeWeather = () => {
      const isHot = this.state.isHot
      this.setState({ isHot: !isHot })
    }
  }
  ReactDOM.render(<Weather />, document.getElementById('root'))
</script>
```

2.2.2. 理解

1. state 是组件对象最重要的属性，值是对象（可以包含多个 key-value 的组合）
2. 组件被称为"状态机"，通过更新组件的 state 来更新对应的页面显示（重新渲染(render)组件）

2.2.3. 强烈注意：

1. 组件中 render 方法中的 this 为组件实例对象
2. 组件自定义的方法中 this 为 undefined，如何解决?
   - 强制绑定 this：通过函数对象的 bind()

```
constructor(props) {
    console.log('constructor')
    // 构造器调用几次？ -- 1次
    super(props)
    // 初始化状态
    this.state = { isHot: false, wind: '微风' }
    // 解决changeWeather中this指向问题
    // 后面表示顺着weather的原型链找到的changeWeather再使用bind改变this指向，之后赋值挂在自身身上一个changeWeather
    this.changeWeather = this.changeWeather.bind(this)
  }
```

   - 箭头函数

```
// 组件中自定义方法-----要用赋值语句的形式+箭头函数
changeWeather = () => {
  const isHot = this.state.isHot
  this.setState({ isHot: !isHot })
}
```

3. 状态(state)不可直接更改，下面这行就是直接更改！！！

   // this.state.isHot = !isHot // 这是错误的写法

   状态必须通过setState进行更新，且更新是一种合并，不是替换（状态对象中没有更改的保持不变直接使用）

   this.setState({})

2.3. 组件实例三大核心属性 2：props

2.3.1. 效果

需求：自定义用来显示一个人员信息的组件

1. 姓名必须指定，且为字符串类型
2. 性别为字符串类型，如果性别没有指定，默认为男
3. 年龄为字符串类型，且为数字类型，默认值为 18

代码实现：

```
<!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>
  </head>
  <body>
    <!-- 准备好一个“容器” -->
    <div id="root1"></div>
    <div id="root2"></div>
    <div id="root3"></div>

    <!-- 引入react核心库 -->
    <script type="text/javascript" src="../js/react.development.js">
</script>
    <!-- 引入react-dom，用于支持react操作DOM -->
```

```
    <script type="text/javascript" src="../js/react-dom.development.js">
</script>
    <!-- 引入babel，用于将jsx转为js -->
    <script type="text/javascript" src="../js/babel.min.js"></script>

    <script type="text/babel">
      // 创建组件
      class Person extends React.Component {
        state = { name: 'tom', age: 18, sex: '女' }
        render() {
          const {name, age, sex} = this.props
          return (
            <ul>
              <li>姓名: {name}</li>
              <li>性别: {age}</li>
              <li>年龄: {sex}</li>
            </ul>
          )
        }
      }
      // 渲染组件到页面
      ReactDOM.render(<Person name="tom" age="18" sex="女"/>,
document.getElementById('root1'))
      ReactDOM.render(<Person name="tom" age="15" sex="男"/>,
document.getElementById('root2'))
      ReactDOM.render(<Person name="tom" age="19" sex="男"/>,
document.getElementById('root3'))
    </script>
  </body>
</html>
```

2.3.2. 理解

1. 每个组件对象都会有 props(properties 的简写)属性
2. 组件标签的所有属性都保存在 props 中
3. props是只读的

2.3.3. 作用

1. 通过标签属性从组件外向组件内传递变化的数据
2. 注意：组件内部不要修改 props 数据

2.3.4. 编码操作

1. 内部读取某个属性值

```
this.props.name
```

2. 对 props 中的属性进行类型限制和非必要性限制
    - 第一种方式（React v15.5开始弃用）：

```
Person.propType = {
  name: React.PropTypes.string.isRequired,
  age: React.PropTypes.number
}
```

- 第二种方式（新）：使用 prop-types 库进行限制（需要引入 prop-type 库）

```
Person.propType = {
  name: PropType.string.isRequired,
  age: PropType.number
}
```

代码示例（第二种方式）：

```
<!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>
  </head>
  <body>
    <!-- 准备好一个“容器” -->
    <div id="root1"></div>
    <div id="root2"></div>
    <div id="root3"></div>

    <!-- 引入react核心库 -->
    <script type="text/javascript" src="../js/react.development.js"></script>
    <!-- 引入react-dom，用于支持react操作DOM -->
    <script type="text/javascript" src="../js/react-dom.development.js"></script>
    <!-- 引入babel，用于将jsx转为js -->
    <script type="text/javascript" src="../js/babel.min.js"></script>
    <!-- 引入prop-type，用于对组件标签属性进行限制 -->
    <script type="text/javascript" src="./js/prop-types.js"></script>

    <script type="text/babel">
      // 创建组件
      class Person extends React.Component {
        state = { name: 'tom', age: 18, sex: '女' }
        render() {
          const { name, age, sex } = this.props
          return (
            <ul>
              <li>姓名: {name}</li>
              <li>性别: {age + 1}</li>
              <li>年龄: {sex}</li>
            </ul>
          )
        }
      }

      // 对标签属性进行类型和必要性的限制
      Person.propTypes = {
        name: PropTypes.string.isRequired, // 限制name必传，且为字符串
        age: PropTypes.number, // 限制age为数值
        sex: PropTypes.string, // 限制sex为字符串
        speak: PropTypes.func // 限制speak为函数
```

```
      }
      // 指定默认的标签属性值
      Person.defaultProps = {
        sex: '男', // 性别默认值为 '男'
        age: 19 // age 默认值为19
      }

      // 渲染组件到页面
      ReactDOM.render(<Person name="tom" age={18} sex="女" speak={speak}
/>, document.getElementById('root1'))
      ReactDOM.render(<Person name="jack" />,
document.getElementById('root3'))

      const p = { name: 'jerry', age: 18 }
      ReactDOM.render(<Person {...p} />,
document.getElementById('root2')) // 批量传递属性(props)
      function speak() {}
    </script>
  </body>
</html>

```

3. 扩展属性：将对象的所有属性通过 props 传递（**批量传递props**）

```
<Person {...Person}/>
```

扩展运算符：

```
// n个数求和
function sum(...num) {
  return numbers.reduce((preValue, currentValue) => {
    return proValue + currentValue
  })
}
console.log(sum(1, 2, 2, 4))

// 构造字面量对象时使用展开语法
var obj1 = { a: 'bar', b: 1 }
var obj2 = { a: 'baz', b: 2 }
var cloneObj = { ...obj1 } // 克隆obj1对象 => 深克隆，不会互相影响

// 合并
let obj3 = {...obj1, a: 'ddd', c: 'aaa'} // 克隆并修改属性

```

4. 设置默认属性值：

```
Person.defaultProps = {
  age: 18,
  sex: '男''
}
```

5. 组件类的构造函数：

   - 通常在 React 中，构造函数仅用于以下两种情况：

1. 通过 `this.state` 赋值对象来初始化内部 state。
2. 为事件处理函数绑定实例

- 构造器是否接收 props，是否传递给 super，取决于：是否希望在构造器函数中通过 this 访问 props

```
constructor(props){
  super(props)
  console.log(props) // 打印所有属性
}
```

6. props的简写方式

```
class Person extends React.Component {
  // 对标签属性进行类型和必要性的限制
  static propTypes = {
    name: PropTypes.string.isRequired, // 限制name必传，且为字符串
    age: PropTypes.number, // 限制age为数值
    sex: PropTypes.string, // 限制sex为字符串
    speak: PropTypes.func, // 限制speak为函数
  }
  // 指定默认的标签属性值
  static defaultProps = {
    sex: '男', // 性别默认值为 '男'
    age: 19, // age 默认值为19
  }
  render() {
    const { name, age, sex } = this.props
    // props是只读的
    // this.props.name = 'jack' // 此行代码会报错，因为props是只读的
    return (
      <ul>
        <li>姓名: {name}</li>
        <li>性别: {age + 1}</li>
        <li>年龄: {sex}</li>
      </ul>
    )
  }
}
```

2.3.5. 函数式组件使用 props

```
<script type="text/babel">
    // 创建组件
    function Person(props) {
      const { name, age, sex } = props
      return (
        <ul>
          <li>姓名: {name}</li>
          <li>性别: {sex}</li>
          <li>年龄: {age}</li>
        </ul>
      )
    }

    // 对标签属性进行类型和必要性的限制
```

```
      Person.propTypes = {
        name: PropTypes.string.isRequired, // 限制name必传，且为字符串
        age: PropTypes.number, // 限制age为数值
        sex: PropTypes.string, // 限制sex为字符串
      }

      // 指定默认的标签属性值
      Person.defaultProps = {
        sex: '男', // 性别默认值为 '男'
        age: 19, // age 默认值为19
      }
      // 渲染组件到页面
      ReactDOM.render(<Person name="tom" age={18} />,
document.getElementById('root1'))
    </script>
```

2.4. 组件实例三大核心属性 3：refs 与事件处理

2.4.1. 效果

需求：自定义组件，功能说明如下：

1. 点击按钮，提示第一个输入框中的值
2. 当第2个输入框失去焦点时，提示这个输入框的值

2.4.2. 理解

组件内的标签可以定义 ref 属性来表示自己

2.4.3. 编码

1. 字符串形式的 ref（已弃用或即将弃用）

```
<input ref="input1"/>
```

2. 回调形式的 ref（尽量避免字符串形式的ref，能不用尽量不用）

```
<input ref={(c) => {this.input1 = c}} />
```

代码示例：

```
<script type="text/babel">
      // 创建组件
      class Demo extends React.Component {
        showData = () => {
          // 展示左侧输入框的数据
          const { input1 } = this
          alert(input1.value)
        }
        showdata2 = () => {
          const { input2 } = this
          alert(input2.value)
        }
        render() {
```

```
          return (
            <div>
              <input
                ref={(currentNode) => (this.input1 = currentNode)}
                type="text"
                placeholder="点击按钮提示数据"
              />
               
              <button onClick={this.showData}>点我提示左侧数据</button>
               
              <input
                onBlur={this.showdata2}
                ref={(currentNode) => (this.input2 = currentNode)}
                type="text"
                placeholder="失去焦点提示数据"
              />
               
            </div>
          )
        }
      }
      // 渲染组件到页面
      ReactDOM.render(<Demo />, document.getElementById('root'))
    </script>
```

回调 ref 中回调执行次数的问题：

- 如果 ref 回调函数是以内联函数的方式定义的，在更新过程中它会被执行两次，第一次传入参数 null，然后第二次传入参数 DOM 元素。这是因为在每次渲染时会创建一个新的函数实例，所以 React 清空旧的 ref 并且设置新的。通过 ref 的回调函数定义成 class 的绑定函数的方式可以避免上述问题。但是大多数情况下它是无关紧要的，开发中我们一般就用内联形式

```
<script type="text/babel">
      // 创建组件
      class Demo extends React.Component {
        state = { isHot: true }

        showInfo = () => {
          const { input1 } = this
          alert(input1.value)
        }

        changeWeather = () => {
          // 获取原来的状态
          const { isHot } = this.state
          // 更新状态
          this.setState({ isHot: !isHot })
        }

        saveInput = (c) => {
          this.input1 = c
          console.log('@', c);
        }

        render() {
          const { isHot } = this.state
          return (
```

```
            <div>
              <h2>今天天气很{isHot ? '炎热' : '凉爽'}</h2>
              <br />
              {/*<input
                ref={(c) => {
                  this.input1 = c
                  console.log('@', c)
                }}
                type="text"
              />*/}
              <input ref={this.saveInput} type="text" />
              <button onClick={this.showInfo}>点我提示输入的数据</button>
              <button onClick={this.changeWeather}>点我切换天气</button>
            </div>
          )
        }
      }
      // 渲染组件到页面
      ReactDOM.render(<Demo />, document.getElementById('root'))
    </script>
```

3. createRef 创建 ref 容器

```
myRef = React.createRef()
<input ref={this.myRef} />
```

代码示例：

```
    <script type="text/babel">
      // 创建组件
      class Demo extends React.Component {
        /*
          React.createRef调用后可以返回一个容器，改容器可以存储被ref所标识的节点，该容器是“专人专用”的
        */
        myRef = React.createRef()
        myRef2 = React.createRef()
        showData = () => {
          // 展示左侧输入框的数据
          alert(this.myRef.current.value)
        }
        showData2 = () => {
          alert(this.myRef2.current.value)
        }
        render() {
          return (
            <div>
              <input ref={this.myRef} type="text" placeholder="点击按钮提示数据" />
               
              <button onClick={this.showData}>点我提示左侧数据</button>
               
              <input
                onBlur={this.showData2}
                ref={this.myRef2}
                type="text"
```

```
                placeholder="失去焦点提示数据"
              />
            </div>
          )
        }
      }
      // 渲染组件到页面
      ReactDOM.render(<Demo />, document.getElementById('root'))
    </script>
```

React.createRef 调用后可以返回一个容器，该容器可以存储被ref所标识的节点，该容器是“专人专用”的

2.4.4. 事件处理

1. 通过 onXxx 属性指定事件处理函数（注意大小写）
   - React 使用的是自定义(合成)事件，而不是使用的原生 DOM 事件 —— 为了更好的兼容性
   - React 中的事件是通过事件委托的方式处理的(委托给组件最外层的元素）—— 为了高效
2. 通过 event.target 得到发生事件的 DOM 元素对象 —— 不要过度的使用 ref

   尽量避免 ref 的使用，当发生事件的元素正好是你需要操作的元素的时候，可以用 event.target 获取节点，代替 ref

```
        // 展示右侧输入框的数据
        showData2 = (event) => {
          alert(event.target.value)
        }

        render() {
          return (
            <div>
              <input onBlur={this.showData2} type="text" placeholder="失去焦点提示数据" />
            </div>
          )
        }
```

2.5. 收集表单数据

2.5.1. 效果

需求：定义一个包含表单的组件

输入用户名密码后，点击登录提示输入信息

2.5.2. 理解

包含表单的组件分类

1. 受控组件（受到状态的控制），建议使用受控组件 ==> 相当于 Vue 的双向数据绑定

   定义：页面中所有输入类的 DOM，随着你的输入，就把输入内容维护到状态里面去，等需要用的时候直接从状态里面取出来，这就是属于受控组件

```
    <script type="text/babel">
```

```
      // 创建组件
      class Login extends React.Component {
        // 初始化状态
        state = {
          username: '', // 用户名
          password: '' // 密码
        }

        // 保存用户名到状态中
        saveUsername = (event) => {
          this.setState({username: event.target.value})
        }
        // 保存密码到状态中
        savePassword = (event) => {
          this.setState({password: event.target.value})
        }

        // 表单提交的回调
        handelSubmit = (event) => {
          event.preventDefault()
          const { username, password } = this.state
          alert(`你输入的用户名是：${username}, 你输入的密码是：${password}`)
        }
        render() {
          return (
            <form action="http://www.atguigu.com" onSubmit={this.handelSubmit}>
              用户名：
              <input onChange={this.saveUsername} type="text" name="username" />
              密码：
              <input onChange={this.savePassword} type="password" name="password" />
              <button>登录</button>
            </form>
          )
        }
      }
      // 渲染组件到页面
      ReactDOM.render(<Login />, document.getElementById('root'))
    </script>
```

2. 非受控组件

定义：页面中所有输入类的DOM，是现用现取的，就是非受控组件

```
    <script type="text/babel">
      // 创建组件
      class Login extends React.Component {
        handelSubmit = (event) => {
          event.preventDefault()
          const { username, password } = this
          alert(`你输入的用户名是：${username.value}, 你输入的密码是：${password.value}`)
        }
        render() {
          return (
```

```
            <form action="http://www.atguigu.com" onSubmit={this.handelSubmit}>
              用户名:
              <input ref={c => this.username = c} type="text" name="username" />
              密码:
              <input ref={c => this.password = c} type="password" name="password" />
              <button>登录</button>
            </form>
          )
        }
      }
      // 渲染组件到页面
      ReactDOM.render(<Login />, document.getElementById('root'))
    </script>
```

2.5.3. 高阶函数和函数柯里化

高阶函数：如果一个函数符合下面2个规范中的任何一个，那么该函数就是高阶函数。

1. 若A函数，接受的参数是一个函数，那么A函数就可以称之为高阶函数。
2. 若A函数，调用的返回值依然是一个函数，那么A函数就可以称之为高阶函数。

   常见的高阶函数有：Promise、setTimeout、setInterval、arr.map()等等（数组身上的大部分方法都是高阶函数）

函数柯里化：通过函数调用继续返回函数的方式，实现多次接收参数最后统一处理的函数编码形式。

```
// 函数柯里化
funtion sum(a) {
  return (b) => {
    return (c) => {
      return a + b + c
    }
  }
}
const result = sum(1)(2)(3)
console.log(result)

saveFormData = (dataType) => { // 高阶函数、函数柯里化
  return (event) => {
        // {[dataType]:event.target.value} 表示读取dataType变量作为属性名，值为event.target.value
    this.setState({[dataType]: event.target.value})
  }
}
```

高阶函数和函数柯里化代码示例（柯里化实现受控组件）：

```
    <script type="text/babel">
      // 创建组件
      class Login extends React.Component {
        // 初始化状态
        state = {
          username: '', // 用户名
          password: '', // 密码
```

```
        }

        // 保存表单数据到状态中
        saveFormData = (dataType) => { // 高阶函数、函数柯里化
          return (event) => {
            // {[dataType]:event.target.value} 表示读取dataType变量作为属性名，值为event.target.value
            this.setState({[dataType]: event.target.value})
          }
        }

        // 表单提交的回调
        handelSubmit = (event) => {
          event.preventDefault()
          const { username, password } = this.state
          alert(`你输入的用户名是：${username}, 你输入的密码是：${password}`)
        }

        render() {
          return (
            <form action="http://www.atguigu.com" onSubmit={this.handelSubmit}>
              用户名：
              <input onChange={this.saveFormData('username')} type="text" name="username" />
              密码：
              <input onChange={this.saveFormData('password')} type="password" name="password" />
              <button>登录</button>
            </form>
          )
        }
      }
      // 渲染组件到页面
      ReactDOM.render(<Login />, document.getElementById('root'))
    </script>
```

不用柯里化实现受控组件：

```
    <script type="text/babel">
      // 创建组件
      class Login extends React.Component {
        // 初始化状态
        state = {
          username: '', // 用户名
          password: '', // 密码
        }

        // 保存表单数据到状态中
        saveFormData = (dataType, event) => {
          this.setState({ [dataType]: event.target.value })
        }

        // 表单提交的回调
        handelSubmit = (event) => {
          event.preventDefault()
          const { username, password } = this.state
```

```
          alert(`你输入的用户名是：${username}，你输入的密码是：${password}`)
        }

        render() {
          return (
            <form action="http://www.atguigu.com" onSubmit=
{this.handelSubmit}>
              用户名：
              <input
                onChange={(event) => this.saveFormData('username', event)}
                type="text"
                name="username"
              />
              密码：
              <input
                onChange={(event) => this.saveFormData('password', event)}
                type="password"
                name="password"
              />
              <button>登录</button>
            </form>
          )
        }
      }
      // 渲染组件到页面
      ReactDOM.render(<Login />, document.getElementById('root'))
    </script>
```

- 类知识复习：[a]表示读取 a 变量（参数）的值

```
<script>
  let a = 'name'
  let obj = {}
  // obj.a = 'tom' // 表示以a作为对象的属性{a: 'tom'}
  obj[a] = 'tom' // 读取a变量的值作为对象的属性{name: 'tom'}
  console.log(obj)
</scrip>
```

2.6. 组件的生命周期

挂载：mount

卸载：unmount

2.6.1. 效果

需求：定义组件实现以下功能；

1. 让指定饿文本做显示 / 隐藏的渐变动画
2. 从完全可见，到彻底消失，耗时 2S
3. 点击 "不活了" 按钮从界面中卸载组件

代码实现（引出生命周期）：

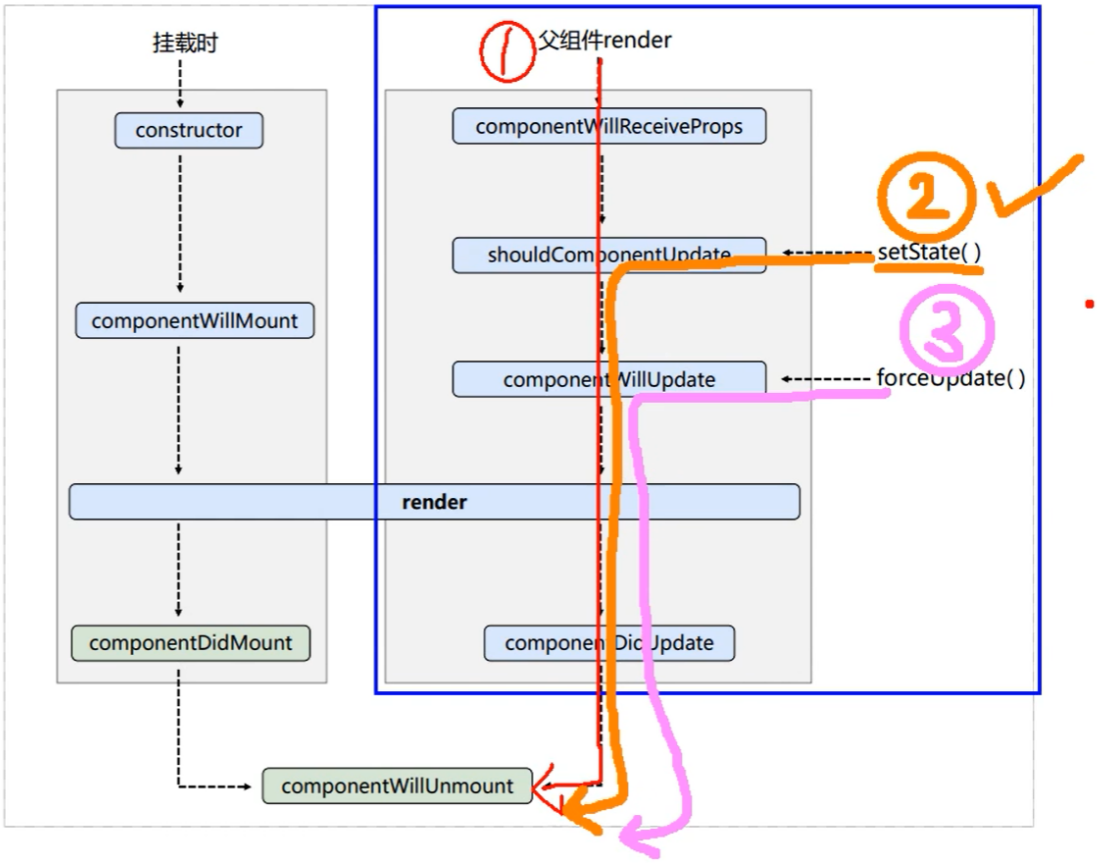
```
<body>
```

```
    <!-- 准备好一个“容器” -->
    <div id="root"></div>

    <!-- 引入react核心库 -->
    <script type="text/javascript" src="../js/react.development.js">
</script>
    <!-- 引入react-dom，用于支持react操作DOM -->
    <script type="text/javascript" src="../js/react-dom.development.js">
</script>
    <!-- 引入babel，用于将jsx转为js -->
    <script type="text/javascript" src="../js/babel.min.js"></script>

    <script type="text/babel">
      // 创建组件
      class Life extends React.Component {
        state = { opacity: 0.1 }

        death = () => {
          // 卸载组件
          // 生命周期回调函数 <=> 生命周期勾子函数 <=> 生命周期函数 <=> 生命周期勾子
          ReactDOM.unmountComponentAtNode(document.getElementById('root'))
        }

        // 组件挂载完毕之后调用
        componentDidMount() {
          this.timer = setInterval(() => {
            // 获取原状态
            let { opacity } = this.state
            opacity -= 0.1
            if (opacity <= 0) opacity = 1
            // 设置新的透明度
            this.setState({ opacity })
          }, 200)
        }

        // 组件将要卸载时调用
        componentWillUnmount() {
          // 清除定时器
        clearInterval(this.timer)
        }

        // render调用的时机：初始化渲染，状态更新之后
        render() {
          console.log('render')
          return (
            <div>
              <h2 style={{ opacity: this.state.opacity }}>React学不会怎么办？
</h2>
              <button onClick={this.death}>不活了</button>
            </div>
          )
        }
      }
      // 渲染组件到页面
      ReactDOM.render(<Life />, document.getElementById('root'))
    </script>
  </body>
```

2.6.2. 理解

1. 组件从创建到死亡它会经历一个特定的阶段。
2. React 组件中包含一系列勾子函数（生命周期回调函数），会在特定的时刻调用。
3. 我们在定义组件时，会在特定的生命周期回调函数中，做特定的工作。

2.6.3. 生命周期流程图(旧)



生命周期的三个阶段（旧）

1. **初始化阶段：**由 ReactDOM.render() 触发 —— 初次渲染
    1. constructor()
    2. componentWillMount()
    3. render()
    4. **componentDidMount() =====> 常用**
        - 一般在这个勾子中做一些初始化的事，例如：开启定时器、发送网络请求、订阅消息
2. **更新阶段：**由组件内部 this.setState() 或父组件重新 render 触发（强制更新 this.forceUpdate，就是少一个环节，不受阀门控制）
    1. shouldComponentUpdate()
    2. componentWillUpdate()
    3. **render() =====> 必须使用的一个**
    4. componentDidUpdate()
3. **卸载组件：**由 ReactDOM.unmountComponentAtNode() 触发
    1. **componentWillUnmount() =====> 常用**
        - 一般在这个勾子种做一些收尾的工作，例如：关闭定时器、取消订阅消息

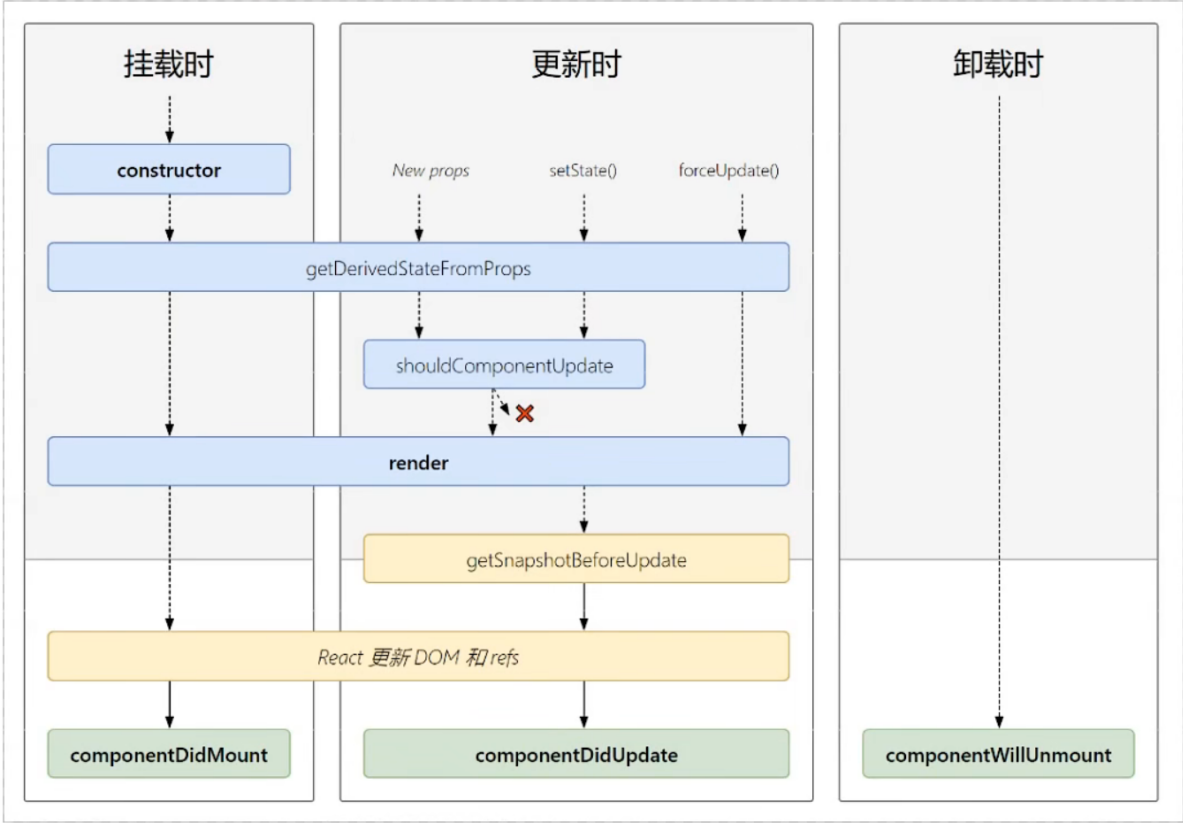
生命周期代码示例：

```html
<!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>
  </head>
  <body>
    <!-- 准备好一个“容器” -->
    <div id="root"></div>

    <!-- 引入react核心库 -->
    <script type="text/javascript" src="../js/react.development.js"></script>
    <!-- 引入react-dom，用于支持react操作DOM -->
    <script type="text/javascript" src="../js/react-dom.development.js"></script>
    <!-- 引入babel，用于将jsx转为js -->
    <script type="text/javascript" src="../js/babel.min.js"></script>

    <script type="text/babel">
      // 创建组件
      class Count extends React.Component {
        // 构造器
        constructor(props) {
          console.log('Count---constructor')
          super(props)
          // 初始化状态
          this.state = { count: 0 }
        }

        // 加1按钮的回调
        add = () => {
          // 获取原状态
          const { count } = this.state
          this.setState({ count: count + 1 })
        }

        // 卸载组件按钮的回调
        death = () => {
          ReactDOM.unmountComponentAtNode(document.getElementById('root'))
        }

        // 强制更新按钮的回调
        force = () => {
          this.forceUpdate()
        }

        // 组件将要挂载的勾子
        componentWillMount() {
          console.log('Count---componentWillMount')
        }

        // 组件挂载完毕的勾子
        componentDidMount() {
          console.log('Count---componentDidMount')
```

```
        }

        // 组件将要卸载的勾子
        componentWillUnmount() {
          console.log('Count---componentWillUnmount')
        }

        // 控制组件更新的阀门
        // 该勾子如果不写，底层也会给你补一个，且默认返回值为真；如果写了，则必须写一个返回值（true / false）
        shouldComponentUpdate() {
          console.log('Count---shoudComponentUpdate')
          return true
        }

        // 组件将要更新的勾子
        componentWillUpdate() {
          console.log('Count---componentWillUpdate')
        }

        // 组件更新完毕的勾子
        componentDidUpdate() {
          console.log('Count---componentWillUpdate')
        }

        render() {
          console.log('Count---render')
          const { count } = this.state
          return (
            <div>
              <h2>当前求和为{count}</h2>
              <button onClick={this.add}>点我+1</button>
              <button onClick={this.death}>卸载组件</button>
              <button onClick={this.force}>不更改任何状态中的数据，强制更新</button>
            </div>
          )
        }
      }

      // 父组件A
      class A extends React.Component {
        // 初始化状态
        state = { carName: '丰田' }

        changeCar = () => {
          this.setState({ carName: '红旗HS9' })
        }

        render() {
          const { carName } = this.state
          return (
            <div>
              <div>我是A组件</div>
              <button onClick={this.changeCar}>换车</button>
              <B carName={carName} />
            </div>
          )
```

```
        }
      }

      // 子组件B
      class B extends React.Component {
        // 组件将要接收新的props的勾子
        componentWillReceiveProps(props) {
          // 注意：第一次接受的不算，第一次接收props不会调用
componentWillReceiveProps
          console.log('B---componentWillReceiveProps', props)
        }
        // 控制组件更新的阀门
        shouldComponentUpdate() {
          console.log('B---shouldComponentUpdate')
          return true
        }
        // 组件将要更新的勾子
        componentWillUpdate() {
          console.log('B---componentWillUpdate')
        }
        // 组件更新完成的勾子
        componentDidUpdate() {
          console.log('B---componentDidUpdate')
        }
        render() {
          console.log('B---render')
          return <div>我是B组件,接收到的车是:{this.props.carName}</div>
        }
      }

      // 渲染组件到页面
      ReactDOM.render(<Count />, document.getElementById('root'))
    </script>
  </body>
</html>

```

2.6.4. 生命周期流程图（新）



生命周期的三个阶段（新）

1. **初始化阶段：**由ReactDOM.render() 触发——初次渲染
    1. constructor()
    2. getDerivedStateFromProps
        - 若你的 state 的值在任何时候都取决于 props的时候，可以使用 。需要返回一个状态对象或者null
    3. render()
    4. **componentDidMount() =====> 常用**
        - 一般在这个勾子中做一些初始化的事，例如：开启定时器、发送网络请求、订阅消息
2. **更新阶段：**由组件内部 this.setState() 或父组件重新 render 触发
    1. getDerivedStateFromProps
    2. shouldComponentUpdate()
    3. **render()**
    4. getSnapshotBeforeUpdate
        - getSnashotBeforeUpdate() 在最近一次渲染输出（提交到 DOM 节点）之前调用。它使得组件能在发生更改之前从 DOM 中捕获一些信息（例如，滚动位置）。此生命周期的任何返回值都将作为参数传递给 componentDidUpdate()
        - 此用法并不常见，但它可能出现在 UI 处理中，如需要以特殊方式处理滚动位置的聊天线程等。
    5. componentDidUpdate()
3. **卸载组件：**由 ReactDOM.unmountComponentAtNode() 触发
    1. **componentWillUnmount() =====> 常用**
        - 一般在这个勾子种做一些收尾的工作，例如：关闭定时器、取消订阅消息

生命周期（新）代码示例：

```
<!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生命周期</title>
  </head>
  <body>
    <!-- 准备好一个“容器” -->
    <div id="root"></div>

    <!-- 引入react核心库 -->
    <script type="text/javascript" src="./js/17.0.1/react.development.js"></script>
    <!-- 引入react-dom，用于支持react操作DOM -->
    <script type="text/javascript" src="./js/17.0.1/react-dom.development.js"></script>
    <!-- 引入babel，用于将jsx转为js -->
    <script type="text/javascript" src="./js/17.0.1/babel.min.js"></script>

    <script type="text/babel">
      // 创建组件
      class Count extends React.Component {
        // 构造器
        constructor(props) {
          console.log('Count---constructor')
          super(props)
          // 初始化状态
          this.state = { count: 0 }
        }

        // 加1按钮的回调
        add = () => {
          // 获取原状态
          const { count } = this.state
          this.setState({ count: count + 1 })
        }

        // 卸载组件按钮的回调
        death = () => {
          ReactDOM.unmountComponentAtNode(document.getElementById('root'))
        }

        // 强制更新按钮的回调
        force = () => {
          this.forceUpdate()
        }

        // 若你的 state 的值在任何时候都取决于 props的时候，可以使用getDerivedStateFromProps，需要返回一个状态对象或者null
        static getDerivedStateFromProps(props, state) {
          console.log('getDerivedStateFromProps', props, state)
          return null
        }

// 在更新之前获取快照，返回出去传给 componentDidUpdate
```

```
        getSnapshotBeforeUpdate() {
          console.log('getSnapshotBeforUpdate')
          return 'bilibili'
        }

        // 组件挂载完毕的钩子
        componentDidMount() {
          console.log('Count---componentDidMount')
        }

        // 组件将要卸载的钩子
        componentWillUnmount() {
          console.log('Count---componentWillUnmount')
        }

        // 控制组件更新的阀门
        // 该钩子如果不写，底层也会给你补一个，且默认返回值为真；如果写了，则必须写一个返回值（true / false）
        shouldComponentUpdate() {
          console.log('Count---shoudComponentUpdate')
          return true
        }

        // 组件更新完毕的钩子
        componentDidUpdate(preprops, prestate, snapshotValue) {
          console.log('Count---componentDidUpdate', preprops, prestate, snapshotValue)
        }

        render() {
          console.log('Count---render')
          const { count } = this.state
          return (
            <div>
              <h2>当前求和为{count}</h2>
              <button onClick={this.add}>点我+1</button>
              <button onClick={this.death}>卸载组件</button>
              <button onClick={this.force}>不更改任何状态中的数据，强制更新</button>
            </div>
          )
        }
      }

      // 渲染组件到页面
      ReactDOM.render(<Count count={199}/>, document.getElementById('root'))
    </script>
  </body>
</html>
```

getSnapshotBeforeUpdate 勾子使用场景：（状态更新后需要获取之前的状态时使用）

```
    <script type="text/babel">
      // 创建组件
      class NewsList extends React.Component {
```

```
        state = { newsArr: [] }

        componentDidMount() {
          setInterval(() => {
            // 获取原状态
            const { newsArr } = this.state
            // 模拟一条新闻
            const news = '新闻' + (newsArr.length + 1)
            // 更新状态
            this.setState({newsArr: [news, ...newsArr]})
          }, 1000)
        }

        getSnapshotBeforeUpdate() {
          return this.refs.list.scrollHeight
        }

        componentDidUpdate(preProps, preState, height) {
          this.refs.list.scrollTop += this.refs.list.scrollHeight - height
        }
        render() {
          const { newsArr } = this.state
          return (
            <div>
              <div ref="list" className="list">
                {
                  this.state.newsArr.map((n, index) => {
                    return <div key={index} className="news">{n}</div>
                  })
                }
              </div>
            </div>
          )
        }
      }
      // 渲染组件到页面
      ReactDOM.render(<NewsList />, document.getElementById('root'))
    </script>
```

2.6.5. 重要的勾子

1. render：初始化渲染或更新渲染调用
2. componentDidMount：可以在里面开启定时器，开启监听（订阅消息），发送 ajax 请求
3. componentWillUnmount：做一些首位工作，如：清理定时器，取消消息订阅

2.6.6. 即将废弃的勾子

1. componentWillMount
2. componentWillReceiveProps
3. componentWillUpdate

现在使用会出现警告，下一个版本加上 `UNSAFE_` 前缀才能使用，以后可能会被彻底废弃，不建议使用。

2.7. 虚拟 DOM 与 DOM Diffing 算法

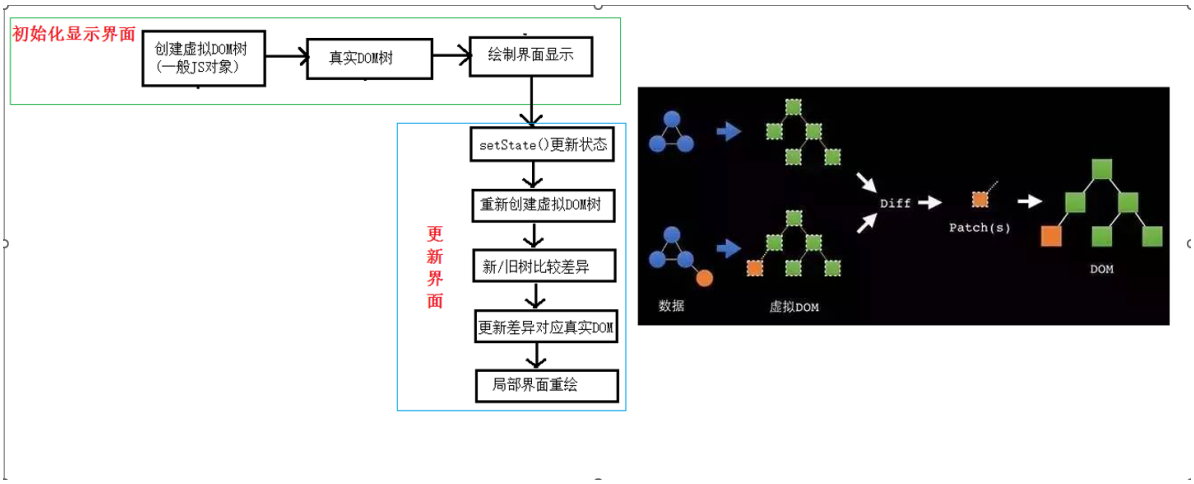
2.7.1. 效果

需求：验证虚拟 DOM Diffing 算法的存在

代码实现：

```
<script type="text/babel">
    // 创建组件
    class Time extends React.Component {
      state = { date: new Date() }

      componentDidMount() {
        setInterval(() => {
          this.setState({
            date: new Date()
          })
        }, 1000)
      }
      render() {
        return (
          <div>
            <h1>hello</h1>
            <input type="text" />
            <span>现在是：{this.state.date.toTimeString()}</span>
          </div>
        )
      }
    }
    // 渲染组件到页面
    ReactDOM.render(<Time />, document.getElementById('root'))
  </script>
```

2.7.2. 基本原理图



2.7.3. key 的作用

经典面试题：

```
1. react / vue 中的 key 有什么作用？（key 的内部原理是什么？）
1. 为什么遍历列表是，key 最好不要用 index？
```

1. 虚拟 DOM 中 key 的作用：
    1. 简单地说：key 是虚拟 DOM 对象的标识，在更新显示时 key 起着极其重要的作用。
    2. 详细的说：当状态发生变化时，react 会根据 【新数据】生成【新的虚拟 DOM】，随后 React 进行【新虚拟 DOM】与【旧虚拟 DOM】的 diff 比较，比较规则如下：

       a. 旧虚拟 DOM 中找到了与新虚拟 DOM 相同的 key：

          (1). 若虚拟 DOM 中 内容没变，直接使用之前的真实 DOM

          (2). 若虚拟 DOM 中的内容改变了，则生成新的真实 DOM，随后替换掉页面之前的真实 DOM

       b.旧虚拟 DOM 中未找到与新虚拟 DOM 相同的 key

          根据数据创建新的真实 DOM，随后渲染到页面

2. 用 index 作为 key 可能会引发的问题：
    1. 若对数据进行：逆序添加、逆序删除等破坏顺序的操作：

       会产生没有必要的真实 DOM 更新 ==> 页面效果没问题，但效率低。

    2. 如果结构中还包含输入类的 DOM：

       会产生错误 DOM 更新 ===> 页面有问题。

    3. 注意！如果不存在对数据的逆序添加、逆序删除等破坏顺序的操作，仅用于渲染泪飙用于展示，使用 index 作为 key 是没有问题的。

3. 开发中如何选择 key？：
    1. 最好使用每条数据的唯一标识作为 key，比如id、手机号、身份证号、学号等唯一值。
    2. 如果确定只是简单的展示数据，用 index 也是可以的。

代码演示：

```
<!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>jsx小练习</title>
  </head>
  <body>
    <!-- 准备好一个“容器” -->
    <div id="root"></div>

    <!-- 引入react核心库 -->
    <script type="text/javascript" src="./js/react.development.js"></script>
    <!-- 引入react-dom，用于支持react操作DOM -->
    <script type="text/javascript" src="./js/react-dom.development.js"></script>
    <!-- 引入babel，用于将jsx转为js -->
    <script type="text/javascript" src="./js/babel.min.js"></script>
```

```
    <script type="text/babel">
      {/*
        慢动作回放---使用index(索引值)作为key

          初始数据：
              { id: 1, name: '小赵', age: 18 },
              { id: 2, name: '小张', age: 18 }
          初始的虚拟DOM：
              <li key=0>小赵---18<input type="text" /></li>
              <li key=1>小张---18<input type="text" /></li>

          更新后的数据：
              { id: 1, name: '小王', age: 18 },
              { id: 2, name: '小赵', age: 18 },
              { id: 3, name: '小张', age: 20 },
          更新数据后的虚拟DOM：
              <li key=0>小王---20<input type="text" /></li>
              <li key=1>小赵---18<input type="text" /></li>
              <li key=2>小张---18<input type="text" /></li>
----------------------------------------------------------------------
        慢动作回放---使用id(数据的唯一标识)作为key

          初始数据：
              { id: 1, name: '小赵', age: 18 },
              { id: 2, name: '小张', age: 18 }
          初始的虚拟DOM：
              <li key=1>小赵---18<input type="text" /></li>
              <li key=2>小张---18<input type="text" /></li>

          更新后的数据：
              { id: 1, name: '小王', age: 18 },
              { id: 2, name: '小赵', age: 18 },
              { id: 3, name: '小张', age: 20 },
          更新数据后的虚拟DOM：
              <li key=3>小王---20<input type="text" /></li>
              <li key=1>小赵---18<input type="text" /></li>
              <li key=2>小张---18<input type="text" /></li>
      */}

      // 创建组件
      class Person extends React.Component {
        state = {
          persons: [
            { id: 1, name: '小赵', age: 18 },
            { id: 2, name: '小张', age: 18 },
            { id: 3, name: '小李', age: 18 },
          ],
        }

        add = () => {
          const { persons } = this.state
          const p = { id: persons.length + 1, name: '小王', age: 20 }
          this.setState({ persons: [p, ...persons] })
        }
        render() {
          console.log(1)
          return (
```

```
            <div>
              <h2>展示人员信息</h2>
              <button onClick={this.add}>添加一个小王</button>
              <h3>使用index(索引值)作为key</h3>
              <ul>
                {this.state.persons.map((personObj, index) => {
                  return (
                    <li key={index}>
                      {personObj.name}---{personObj.age} <input type="text" />
                    </li>
                  )
                })}
              </ul>
              <hr />
              <hr />
              <h3>使用id(数据的唯一标识)作为key</h3>
              <ul>
                {this.state.persons.map((personObj, index) => {
                  return (
                    <li key={personObj.id}>
                      {personObj.name}---{personObj.age} <input type="text" />
                    </li>
                  )
                })}
              </ul>
            </div>
          )
        }
      }
      // 渲染组件到页面
      ReactDOM.render(<Person />, document.getElementById('root'))
    </script>
  </body>
</html>
```

3. 第三章：React 应用（基于 React 脚手架）

3.1. 使用 create-react-app 创建 react 应用

3.1.1. react 脚手架

1. xxx 脚手架：用来帮助程序员快速创建一个基于 xxx 库的模板项目
    1. 包含了所有需要的配置（语法检查、jsx 编译、devServer…）
    2. 下载好了所有相关的依赖
    3. 可以直接运行一个简单效果
2. react 提供了一个用于创建 react 项目的脚手架库：create-react-app
3. 项目的整体框架为：react + webpack + es6 + eslint

4. 使用脚手架开发的项目的特点：模块化，组件化，工程化（在项目当中使用了类似于webpack这种全自动的构建工具，那么你的项目就是工程化项目，简单点说就是，如果你通过这种构建工具完成了一条龙服务）

3.1.2. 创建项目并启动

第一步，**全局安装**：`npm i -g create-react-app`

第二部，切换到像创建项目的目录，使用命令：`create-react-app hello-react`

第三步，进入项目文件夹：`cd hello-react`

第四步，启动项目：npm start

3.1.3. react脚手架项目结构

public ---- 静态资源文件夹

favicon.icon ------ 网站页签图标

index.html--------主页面

`%PUBLIC_URL%` 代表 public 文件夹的路径

logo192.png ------- logo图

logo512.png ------- logo图

manifest.json ----- 应用加壳的配置文件

robots.txt -------- 爬虫协议文件

src ---- 源码文件夹

App.css -------- App组件的样式

App.js---------App组件

App.test.js ---- 用于给App做测试

index.css ------ 样式

index.js------入口文件

App外侧包裹`<React.StrictMode>`之后能检查App以及其子组件中的写的东西是否合理

logo.svg ------- logo图

reportWebVitals.js

--- 页面性能分析文件(需要web-vitals库的支持)

setupTests.js

---- 组件单元测试的文件(需要jest-dom库的支持)

3.1.4. 样式模块化

解决汇总之后样式名冲突的问题（用less 形成嵌套关系之后一般就不会有问题了）

分以下几步：

1. 样式文件名字改为 `xxx.module.css`，如：index.module.css
2. 引入时：

```
import hello from './index.module.css'
```

3. 类名写为：

```
<h2 className={hello.title}>Welcome</h2>
```

代码示例：

```
import React, { Component } from 'react'
import hello from './index.module.css' // hello 是自己取的

export default class Welcome extends Component {
  render() {
    return <h2 className={hello.title}>Welcome</h2>
  }
}

```

3.1.5. 一个插件的安装（react 代码片段，代码补齐）

ES7+ React/Redux/React-Native snippets

3.1.6. 功能界面的组件化编程流程（通用）

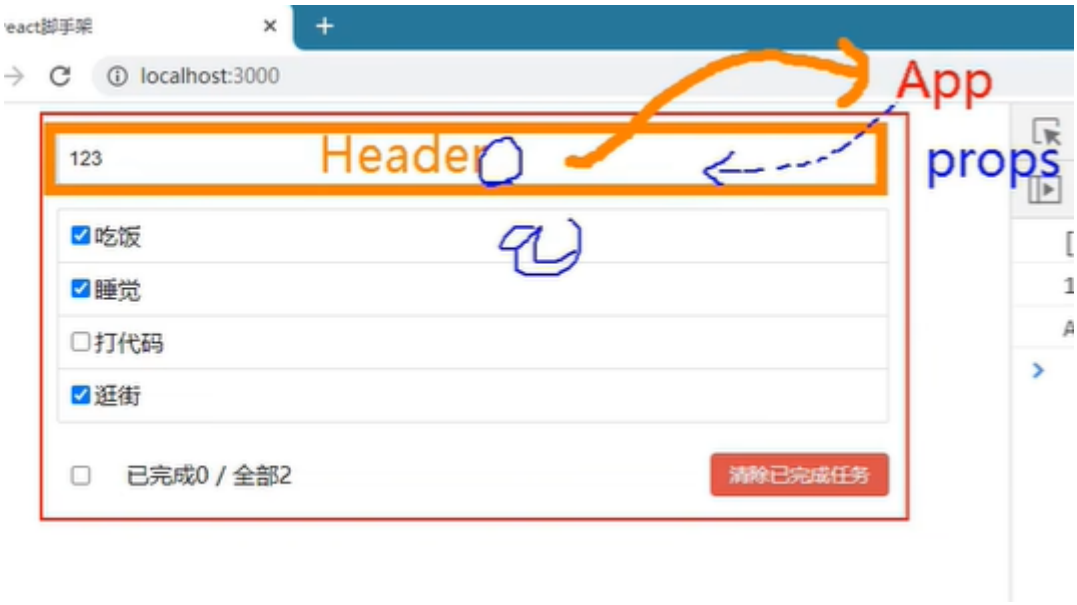
1. 拆分组件：拆分界面，抽取组件
2. 实现静态组件：使用组件实现静态页面效果
3. 实现动态组件
    1. 动态显示初始化数据
        1. 数据类型
        2. 数据名称
        3. 保存在那个组件?
    2. 交互（从绑定事件监听开始）

3.2. 组件的组合使用——TodoList

功能：组件化实现此功能

1. 显示所有 todo 列表
2. 输出文本，点击按钮显示到列表的首位，并清除输入的文本

如果子组件想给父组件传递东西，可以让父组件开始的时候通过props给子组件传递一个函数，子组件在想给父组件传数据的时候调用这个函数把数据传进去，传递给父组件



3.2.1. 一个生成id 的库

1. 下载

```
yarn add nanoid
```

2. 使用

```
import {nanoid} from 'nanoid'

console.log(nanoid()) // nanoid是一个函数，调用会生成一个全球唯一的字符串
```

3.2.2. todoList 知识点总结

1. 拆分组件、实现静态组件，注意：className、style的写法
2. 动态初始化列表，如何确定数据放在哪个组件的 state 中?
   - 某个组件使用：放在其自身的 state 中
   - 某些组件使用：放在他们共同的父组件 state 中（官方称此操作为：状态提升）
3. 关于父子组件之间通信：
   1. 【父组件】给【子组件】传递数据：通过 props 传递
   2. 【子组件】给【父组件】传递数据：通过 props 传递，要求父组件提前给子组件传递一个函数
4. 注意 defaultChecked 和 checked 的区别，类似的还有 defaultValue 和 value
   - input标签的 checked属性要配合onChange函数使用，属性defaultChecked 只会在第一次指定的时候起作用
5. 状态在哪里，操作状态的方法就在哪里
6. `if (window.confirm)` 点击确定返回 `true`，点击取消，返回 `false`。记得前面加 window

4. 第四章：React ajax

4.1. 理解

4.1.1. 前置说明

1. React 本身只关注于界面，并不包含发送 ajax 请求的代码
2. 前端应用需要通过 ajax 请求与后台数据进行交互（json数据）
3. react 应用中需要继承第三方 ajax 库（或自己封装）

4.1.2. 常用的 ajax 请求库

1. jQuery：比较重，如果需要另外引入不建议使用
2. axios：轻量级，建议使用
    1. 封装 XMLHttpRequest 对象的 ajax
    2. promise 风格
    3. 可以用在浏览器端和 node 服务器端

4.2. axios

安装axios：

```
yarn add axios
```

4.2.1. 文档

https://github.com/axios/axios

4.2.2. 相关 API

1. GET 请求

```
axios.get('/user?ID=123')
  .then(function (response) {
    console.log(response.data)
  })
  .catch(function (error) {
    console.log(error)
  })

axios.get('/user', {
  params: {
    ID: 123
  }
  })
  .then(function (response) {
    console.log(response)
  })
  .catch(function (error) {
    console.log(error)
  })
```

2. POST 请求

```
axios.post('/user', {
  firstName: 'Jack',
  lastName: 'Slite'
})
.then(function (response) {
  console.log(response)
})
.catch(function (error) {
  console.log(error)
})
```

4.2.3. react 脚手架配置代理总结

4.2.3.1. 方法一：

在 package.json 中追加如下配置

```
"proxy": "http://localhost:5000"
```

- 上面的配置解决向http://localhost:5000发送请求时产生跨越问题

说明：

1. 优点：配置简单，前端请求资源时可以不加任何前缀。
2. 缺点：不能配置多个代理。
3. 工作方式：上述方式配置代理，当请求了 3000 端口（本地）不存在的资源时，那么该请求会转发给 5000 端口（优先匹配前端资源）

4.2.3.2. 方法二：

1. 第一步：创建代理配置文件

   ```
   在 src 下创建配置文件：src/setupProxy.js
   ```

2. 编写 setupProxy 配置具体代理规则：

   ```
   const { createProxyMiddleware: proxy} = require('http-proxy-middleware')

   module.exports = function(app) {
     app.use(
       proxy('/api1', { // 遇见/api1 时需要转发的请求（所有带有 /api1 前缀的请求都会转发给5000）
         target: 'http://localhost:5000', // 请求配置转发目标地址（能返回数据的服务器地址）
         chageOrigin: true, // 控制服务器接收到的请求头中host字段的值
         /*
           changeOrigin设置为true时，服务器收到的请求头中的host为：localhost:5000
           changeOrigin设置为false时，服务器收到的请求头中的host为：localhost:3000
           changeOrigin默认值为false，但我们一般将changeOrigin值设为true
         */
         pathRewrite: {'^/api1': ''} // 去除请求前缀，保证交给后台服务器的时正常请求地址（必须配置）
       }),
       proxy('/api2', {
   ```

```
      target: 'http://localhost:5000',
      chageOrigin: true,
      pathRewrite: {'^/api2': ''}
    })
  )
}
```

注意：访问地址写代理服务器的地址，拼上 /api1：http://localhost:3000/api1/students

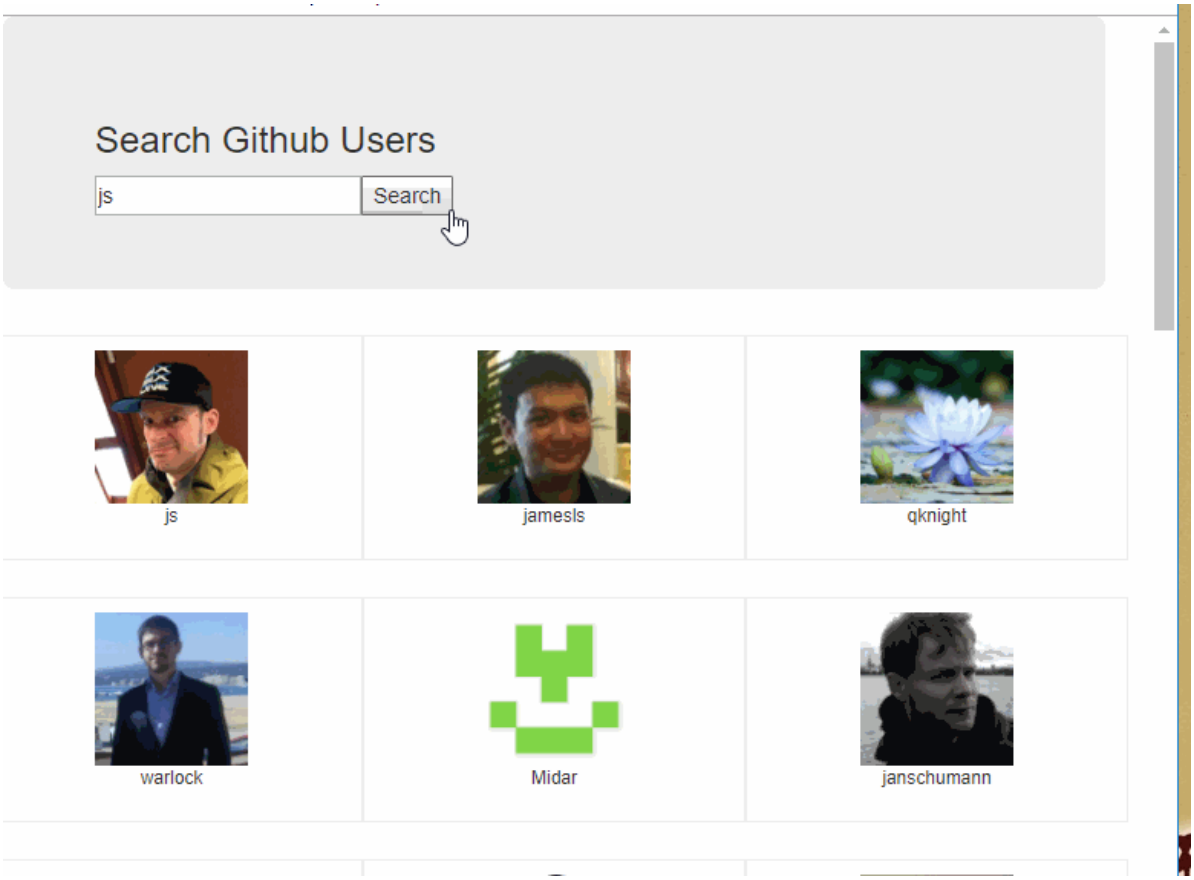
```
import React, { Component } from 'react'
import axios from 'axios'

export default class App extends Component {
  getStudentData = () => {
    axios.get('http://localhost:3000/api1/students').then(
      (response) => {
        console.log('成功了', response.data)
      },
      (error) => {
        console.log('失败了', error)
      }
    )
  }

  render() {
    return (
      <div>
        <button onClick={this.getStudentData}>点我获取学生数据</button>
      </div>
    )
  }
}
```

说明：

1. 优点：可以配置多个代理，可以灵活的控制请求是否走代理。
2. 缺点：配置繁琐，前端请求资源时必须加前缀。

4.3. 案例——github 用户搜索

4.3.1. 效果



请求地址：https://api.github.com/search/users?q=xxxxx

4.3.1.1. 解构赋值连续写法

```
let obj = {a: {b: 1}}
cosnt { a } = obj // 传统解构赋值
const { a: {b}} = obj // 连续解构赋值
const { a: {b: value}} // 连续解构赋值 + 重命名
```

4.3.1.2. 三元表达式可以连续写

```
isFirst ? <h2>欢迎使用，请输入关键字，随后点击搜索</h2> :
        isLoading ? <h2>Loading.....</h2> :
        err ? <h2 style={{color: 'red'}}>{err.message}</h2> :
        this.props.users.map(userObj => {
          return (
            <div key={userObj.id} className="card">
              <a href={userObj.html_url} target="_blank" rel="noreferrer">
                <img alt="head_portrait" src={userObj.avatar_url} style={{
width: '100px' }} />
              </a>
              <p className="card-text">{userObj.login}</p>
            </div>
          )
        })
```

注意：对象不能作为React的节点展示

发请求的位置和请求目标的地址如果一样可以省略，直接写url后边的

4.4. 消息订阅 - 发布机制

1. 工具库：PubSubJS
2. 下载

```
npm install pubsub-js --save
```

3. 使用：
    1. 引入

```
import PubSub from 'pubsub-js' // 引入
```

    2. 订阅

```
PubSub.subscribe('delete', function(data) {}) // 订阅
```

    3. 发布消息

```
PubSub.publish('delete', data) // 发布消息
```

4. 先订阅，再发布（理解：有一种隔空对话的感觉）
5. 适用于任意组件间通信
6. 要在组件 componentWillUnmount 中取消订阅

4.5. 扩展：Fetch

4.5.1. 文档

1. https://github.io/fetch/
2. < https://segmentfault.com/a/1190000003810652>

4.5.2. 特点

1. fetch：原生函数，不再使用 XMLHtmlRequest 对象提交 ajax 请求
2. 老版本浏览器可能不支持

4.5.3. 相关 API

1. GET 请求
    - 未优化

```
// 发送网络请求----使用fetch发送（未优化）
fetch(url).then(
  response => {
    console.log('联系服务器成功了')
    return response.json()
  },
  error => {
    console.log('联系服务器失败了',  error)
    return new Promise()
```

```
  }
).then(
  response => {
    console.log('获取数据成功了', response)
  },
  error => {console.log('获取数据失败了', error)}
)
```

1. GET请求
    - 优化版本

```
fetch(url).then(
  response => {
    console.log('联系服务器成功了')
    return response.json()
  }
).then(
  response => {
    console.log('获取数据成功了', response)
  }
).catch(
  error => {
    console.log('请求出错', error)
  }
)

// 终极优化版本  注意外层函数要加 async
const response = await fetch(url)
const data = response.json()
console.log(data)
```

2. POST 请求

```
fetch(url, {
    method: "POST",
    body: JSON.stringigy(data)
  }).then(function(data) {
    console.log(data)
  }).catch(function(e) {
    console.log(e)
  })
```

4.6. github 搜索案例相关知识点

1. 设计状态时要考虑全面，例如带有网络请求的组件，要考虑请求失败怎么办、第一次加载、加载中、加载成功等情况都要考虑进去。
2. ES6 小知识点：解构赋值 + 重命名

```
let obj = {a: {b: 1}}
cosnt { a } = obj // 传统解构赋值
const { a: {b}} = obj // 连续解构赋值
const { a: {b: value}} // 连续解构赋值 + 重命名
```

3. 消息订阅与发布机制
    1. 先订阅，再发布（理解：有一种隔空对话的感觉）
    2. 适用于任意组件间通信
    3. 要在组件 componentWillUnmount 中取消订阅
4. fetch 发送请求（关注分离的设计思想）

```
try {
  const respons = await fetch(`/api1/search/users?q=${keyWord}`)
  const data = await Response.json()
  PubSub.publish('atguigu', { isLoading: false, users:data.items })
} catch (error) {
    PubSub.publish('atguigu', { isLoading: false, err: error })
}
```

---

5. 第五章：React 路由

5.1. 相关理解

5.1.1. SPA 的理解（单页面、多组件）

1. 单页 Web 应用（single page application，SPA）。
2. 整个应用只有一个完整的页面。
3. 点击页面中的链接不会刷新页面，只会做页面的局部更新。
4. 数据都需要通过 ajax 请求获取，并在前端异步展现。

5.1.2. 路由的理解

1. **什么是路由?**
    1. 一个路由就是一个映射关系（key: value）
    2. key 为路径，value 可能是 function 或 component
2. **路由分类**
    1. 后端路由：
        - 理解：value 是 function，用来处理客户端提交的请求。
        - 注册路由：router.get(path, function(req, res))。
        - 工作过程：当 node 接收到一个请求时，根据请求路径找到匹配的路由，调用路由中的函数来处理请求，返回响应数据。
    2. 前端路由：
        - 浏览器端路由，value 是 component，用于展示页面内容。
        - 注册路由：`<Routr path="/test" component={Test}>`
        - 工作过程：当浏览器的 path 变为 /test 时（点击路由链接时），当前路由组件就会变为 Test 组件

5.1.3. 前端路由原理

- 前端路由借助的是 BOM 上的 history 对象进行工作的
- 需要的话一般借助 history.js 这个库进行操作，

```
  <script type="text/javascript" src="https://cdn.bootcss.com/history/4.7.2/history.js"></script>
  <script type="text/javascript">
    // let history = History.createBrowserHistory() //方法一，直接使用H5推出的history身上的API，兼容性较差
    let history = History.createHashHistory() //方法二，hash值（锚点），几乎没有兼容性问题，兼容性极佳

    function push (path) {
      history.push(path)
      return false
    }

    // 把栈顶的一条记录进行替换，再按回退的话不会回到前一条，而是回到更前一条
    function replace (path) {
      history.replace(path)
    }

    function back() {
      history.goBack()
    }

    function forword() {
      history.goForward()
    }

    history.listen((location) => {
      console.log('请求路由路径变化了', location)
    })
  </script>
```

- 浏览器的历史记录是一个栈解构
- 你只要把 BOM 身上的 history 对象牢牢地握在手里，对浏览器路径以及历史记录的操作你就可以随心所欲了

5.1.4. react-router-dom 的理解

1. react 的一个插件库。
2. 专门用来实现一个 SPA 应用。
3. 基于 react 的项目基本都会用到此库。

5.2. react-router-dom 相关 API

5.2.1. 内置组件

1. `<BrowserRouter>`
2. `<HashRouter>`
3. `<Router>`
4. `<Redirect>`

5. `<Link>`
6. `<NavLink>`
7. `<Switch>`

5.2.2. 其它

1. history 对象
2. match 对象
3. withRouter 函数

5.3. 路由的基本使用

5.3.1. 效果

1. 明确好界面中的导航区、展示区
2. 导航区的 a 标签改为 Link标签，需要高亮的话使用 NavLink 标签

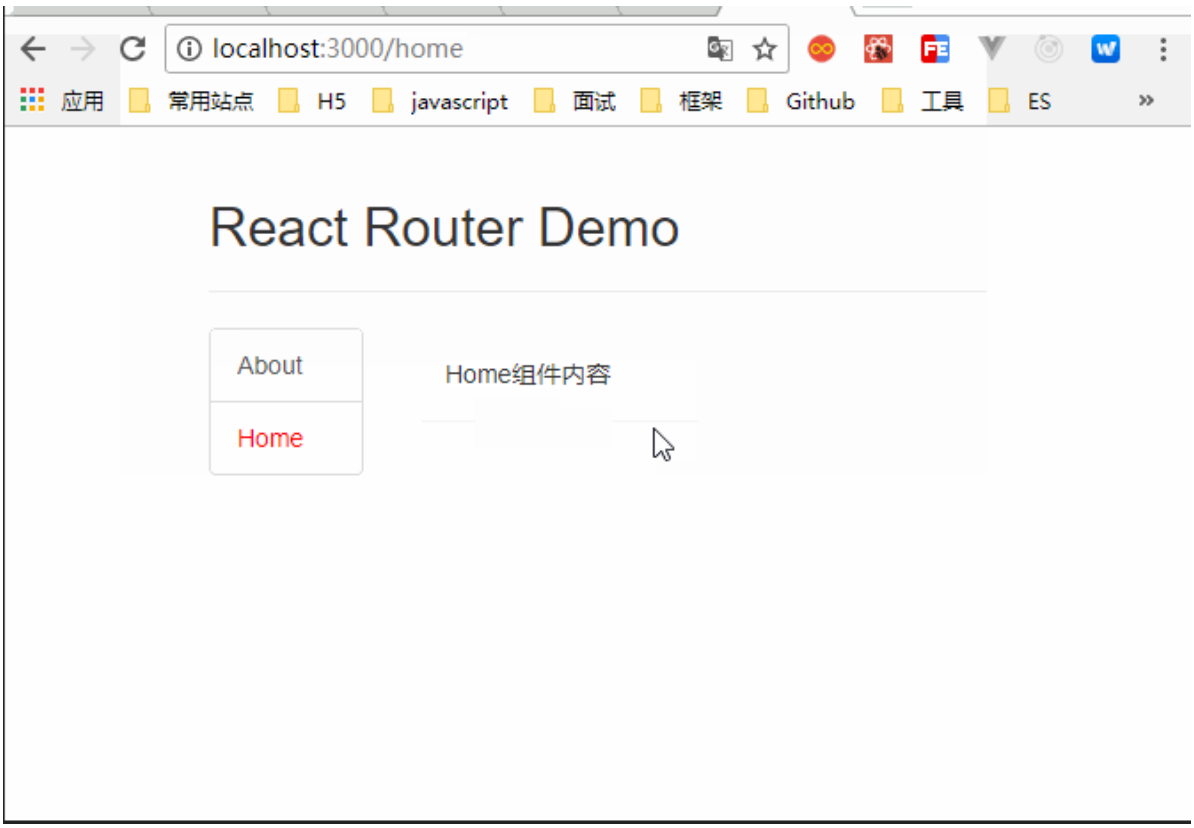
   NavLink 中有一个属性，表示点击时加上属性：activeClassName

```
<Link to="/xxxx" >Demo</Link>

<NavLink activeClassName="atguigu" to="/xxxx" >Demo</NavLink> // 点击时加上类mingatguigu
```

3. 展示区写Router标签进行路径的匹配（注意 component 是小写）

```
<Router path='/xxxx' component={Demo} />
```

4. App 的最外层包裹一个 `<BrowserRouter>` 或 `<HashRouter>`



5.3.2. 准备

1. 下载 react-router-dom

```
npm i --save react-router-dom
```

2. 引入 bootstrap.css

```
<link rel="stylesheet" href="/css/bootstrap.css" >
```

5.4. 路由组件与一般组件的区别

1. 写法不同：
   - 一般组件：`<Demo />`
   - 路由组件：`<Route path="/demo" component={Demo} />`
2. 存放位置不同：
   - 一般组件：components
   - 路由组件：pages
3. 接收到的 props 不同：
   - 一般组件：写组件标签是传递了什么，就能收到什么
   - 路由组件：接收到三个固定的属性
     - **history**:
       1. **go**: *ƒ go(n)*
       2. **goBack**: *ƒ goBack()*
       3. **goForward**: *ƒ goForward()*
       4. **push**: *ƒ push(path, state)*
       5. **replace**: *ƒ replace(path, state)*
     - **location**:
       1. **pathname**: "/home"
       2. **search**: ""
       3. **state**: undefined
     - **match**:
       1. **params**: {}
       2. **path**: "/home"
       3. **url**: "/home"

5.5. NavLink 与封装 NavLink

1. NavLink 可以实现路由链接的高亮（默认会给链接传一个active属性），通过 activeClassName 指定样式名
2. 标签体内容是一个特殊的标签属性：children
3. 通过 this.props.children 可以获取标签体内容

5.5.1. 封装 NavLink

1. 封装 MyNavLink 组件（一般组件）

```
import React, { Component } from 'react'
import { NavLink } from 'react-router-dom'

export default class MyNavLink extends Component {
  render() {
    return (
      // 标签体的内容，传过来是放在this.props.children属性下
      // 公共的属性可以直接放在这里，使用时只需要写不同的地方就好了
      <NavLink activeClassName="atguigu" className="list-group-item"
{...this.props} />
    )
  }
}
```

2. 使用

```
// 标签体内容是一个特殊的标签属性：this.props.children
<MyNavLink to="/about" a={1} b={2}>About</MyNavLink>
<MyNavLink to="/home" a={1} b={2}>Home</MyNavLink>
```

5.6. Switch 的使用

1. 通常情况下，path 和 component 是一一对应的关系。
2. Switch 可以提高路由匹配效率（单一匹配）

```
import { Route, Switch } from 'react-router-dom'

<Switch>
  <Route path="/about" component={About} />
  <Route path="/home" component={Home} />
  <Route path="/home" component={Test} />
</Switch>
```

5.7. 解决多级路径刷新页面样式丢失的问题

1. public/index.html 中 引入样式时不写 ./ 写 /（常用）
2. public/index.html 中 引入样式时不写 ./ 写 %PUBLIC_URL%（常用）
3. 使用 HashRouter

5.8. 路由的严格匹配与模糊匹配

1. 默认使用的是模糊匹配（简单记：【输入的路径(路由连接)】必须包含【匹配的路径(注册路由)】，且顺序要一致）
2. 开启严格匹配：

```
<Routr exact={true} path="/about" component={About} />
```

3. 严格匹配不要顺便开启，必要时再开，有时候开启会导致无法继续匹配二级路由

5.9. Redirect 的使用

1. 一般写在所有路由注册的最下方，当所有路由都无法匹配时，跳转到 Redirect 指定的路由
2. 具体编码：

```
<Switch>
  <Router path="/about" component={About}></Router>
  <Router path="/home" component={Home}></Router>
  <Rediret to="/about" />
</Switch>
```

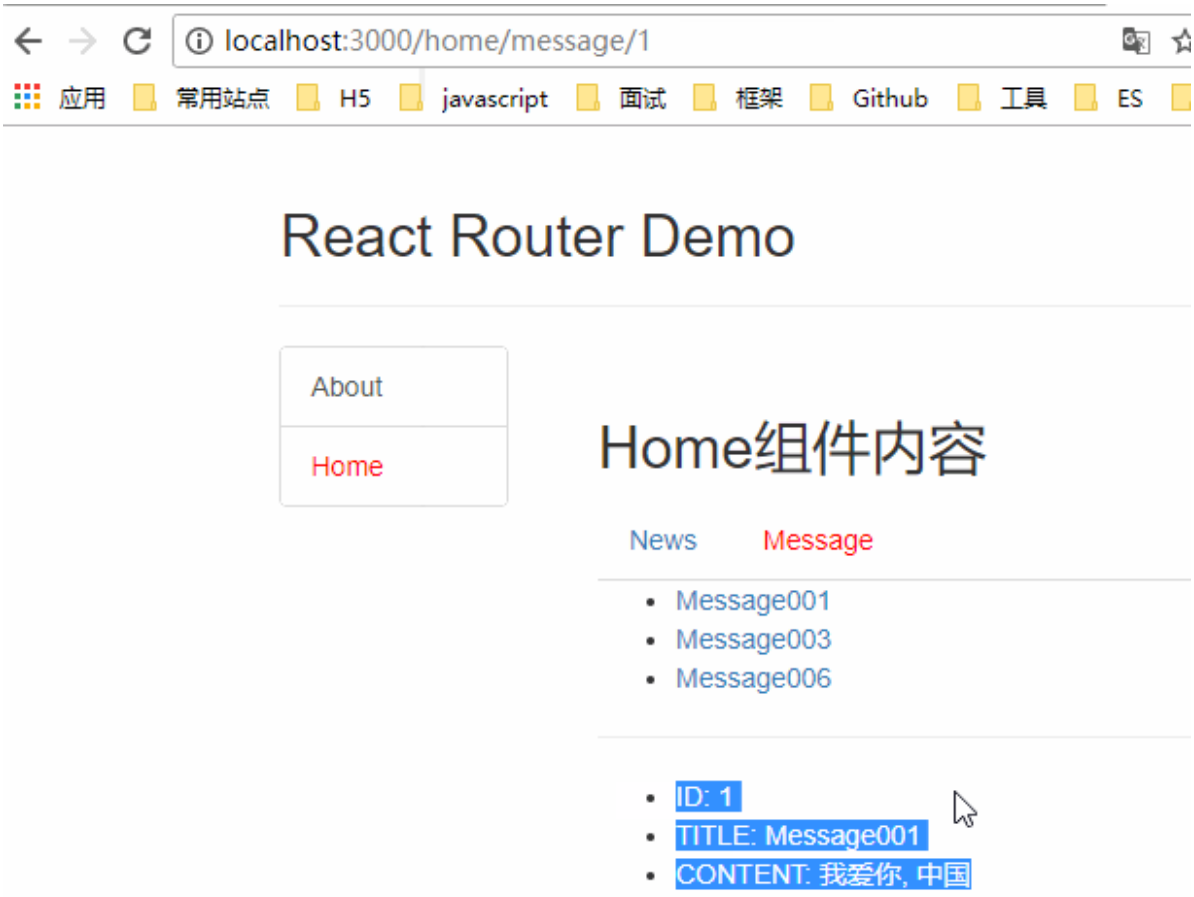
5.10. 嵌套路由使用

1. 组成子路由时要写上父路由的path值：/home/news
2. 路由的匹配是按照注册路由的顺序进行的

5.10.1. 效果

实现步骤：

1. 点击导航链接引起路径变化
2. 路径变化被前端路由器监测到，进行匹配组件，从而展示



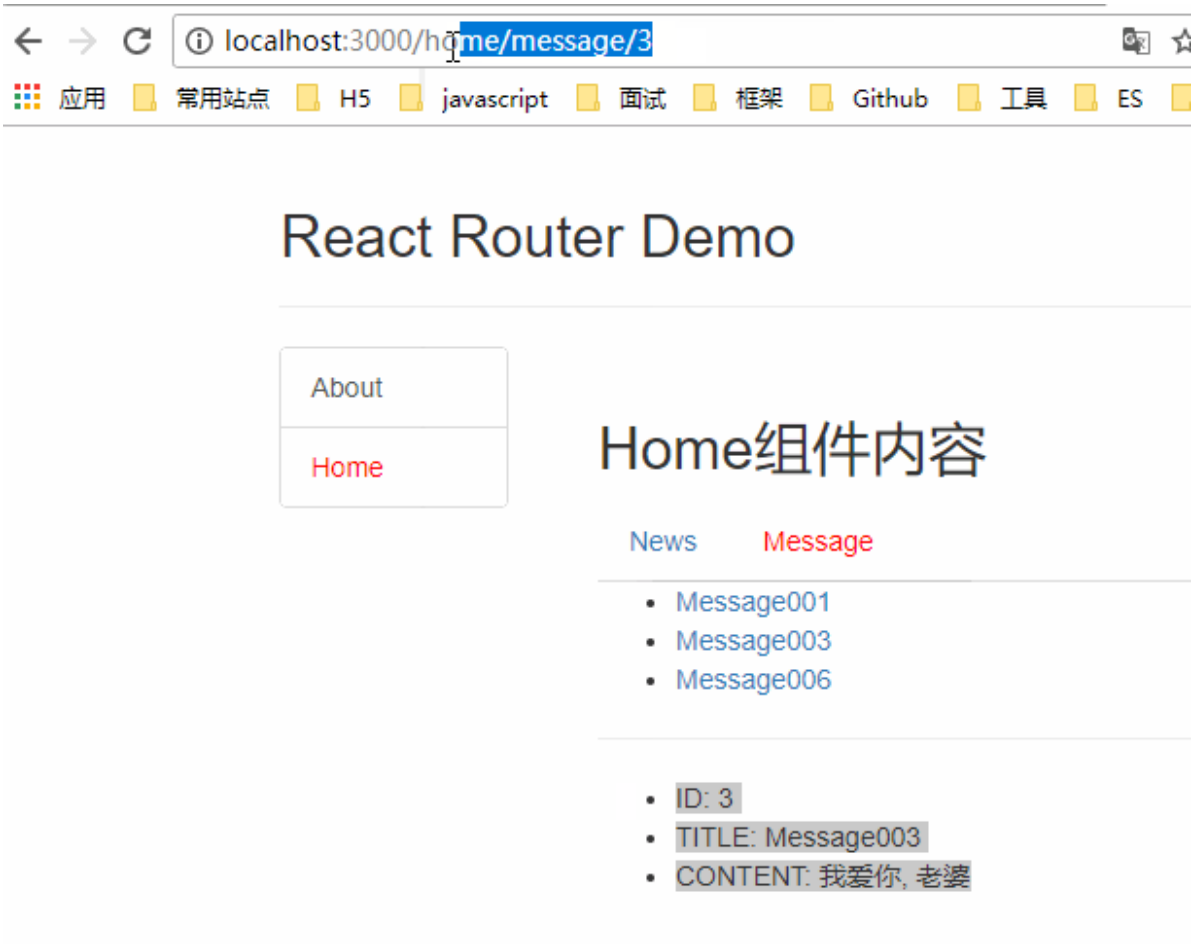
5.11. 向路由组件传递参数数据

params用的最多，其次是search，最后是state，三者都有用，都得掌握

5.11.1. 向路由组件传递params参数

- 路由链接（携带参数）：`Link to="demo/test/tom/18">详情</ Link>`
- 注册路由（声明接收）：`<Route path="demo/test/:name/:age" component={Test} />`
- 接收参数：`const { id, title } = this.props.match.params`

效果：



5.11.2. 向路由组件传递 search 参数

- 路由链接（携带参数）：`Link to="demo/test/?name=tom&age=18">详情</ Link>`
- 注册路由（无需声明，正常注册即可）：`<Route path="demo/test" component={Test} />`
- 接收参数：this.props.location.search

```
import qs from 'query-string' // 需要先下载：yarn add query-string

const { search } = this.props.location
const { id, title } = qs.parse(search.slice(1))
```

- 备注：获取到的search时urlencoded编码字符串，需要借助 query-string 解析

5.11.3. 向路由组件传递state数据

- 路由链接（携带参数）：`Link to={{pathname:'demo/test', state:{name:'tom',age:18}}>详情</ Link>`
- 注册路由（无需声明，正常注册即可）：`<Route path="demo/test" component={Test} />`
- 接收参数：this.props.location.state

```
const { state } = this.props.location
const { id, title } = state
```

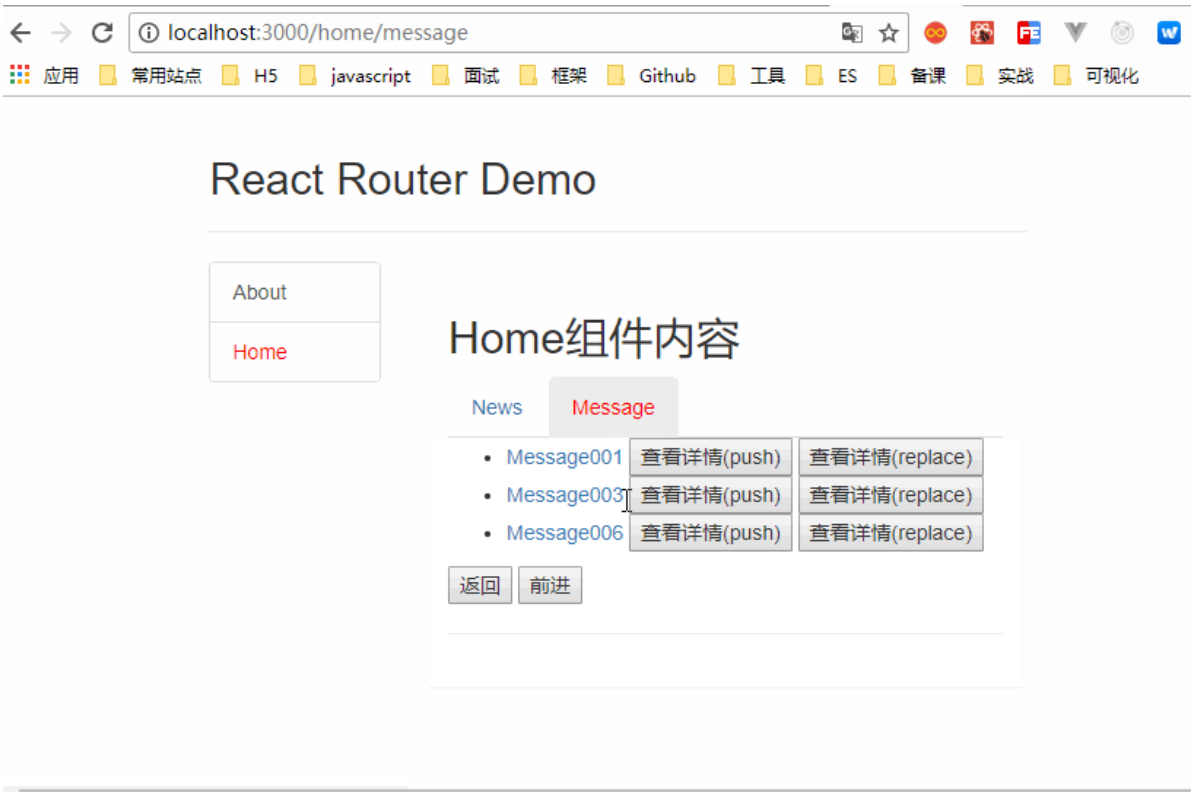
- 备注：刷新也可以保留住参数

5.12. 多种路由跳转方式

直接在路由链接中加属性：replace={true} 或 push={true}，也可以用编程式路由导航实现

- push：跳转后记录会放在历史记录栈顶，可以回退到上一步
- replace：跳转后替换掉栈顶的历史记录，回退会到上上一步

5.12.1. 效果



5.13. 编程式路由导航

- 借助 `this.props.history` 对象上的 API 对路由操作进行跳转、前进、后退
  - this.props.history.push()
  - this.props.history.replace()
  - this.props.history.goBack()
  - this.props.history.goForward()
  - this.props.history.go()

5.14. withRouter 的使用

- withRouter可以加工一般组件，让一般组件具备路由组件所特有的API
- withRouter是一个函数，它的返回值是一个新组件

```
import React, { Component } from 'react'
import { withRouter } from 'react-router-dom'

class Header extends Component {
  back = () => {
    this.props.history.goBack()
  }

  forward = () => {
    this.props.history.goForward()
  }

  go = (number) => {
    this.props.history.go(number)
  }
  render() {
    // console.log('Header组件收到的props是', this.props)
    return (
      <div className="page-header">
        <h2>React Router Demo</h2>
        <button onClick={this.back}>回退</button> 
        <button onClick={this.forward}>前进</button> 
        <button
          onClick={() => {
            this.go(-2)
          }}
        >
          跳转
        </button>
      </div>
    )
  }
}

// withRouter可以加工一般组件，让一般组件具备路由组件所特有的API
// withRouter的返回值是一个新组件
export default withRouter(Header)


```

5.15. BrowserRouter与HashRouter的区别

1. 底层原理不一样；
    - BrowserRouter 使用的是H5的history API，不兼容IE9以下版本。
    - HashRouter 使用的是URL的哈希值。
2. path 表现形式不一样
    - BrowserRouter 的路径中没有#，例如：localhost:3000/demo/test
    - HashRouter 的路径包含#，例如：localhost:3000/#/demo/test

3. 刷新后对路由state参数的影响
    - BrowserRouter 没有任何影响，因为 state 保存在 history 对象中
    - HashRouter 刷新后会导致路由 state 参数的丢失
4. 备注：HashRouter 可以用于解决一些路径错误相关的问题（例如前面的样式丢失异常）。

---

6. 第六章：React UI 组件库

6.1. 流行的开源 React UI 组件库

1. material-ui(国外)
    1. 官网: http://www.material-ui.com/#/
    2. github: https://github.com/callemall/material-ui
2. ant-design(国内蚂蚁金服)
    1. 官网: https://ant.design/index-cn
    2. Github: https://github.com/ant-design/ant-design/

       antd 比较适用于成型的后台管理系统，

       antd使用步骤：
        - 先找东西，看哪个是你想要的，
        - 点击它，然后看代码
        - 代码示例：

```
import React, { Component } from 'react'
import { Button } from 'antd'
import 'antd/dist/antd.min.css'

export default class App extends Component {
  render() {
    return (
      <div>
        App...
        <button>点我</button>
        <Button type="primary">Primary Button</Button>
        <Button >Primary Button</Button>
      </div>
    )
  }
}
```

6.1.1. antd的按需引入+自定义主题

1. 安装依赖：

```
yarn add react-app-rewired customize-cra babel-plugin-import less less-loader
```

2. 修改package.json

```
....
"script": {
  "start": "react-app-rewired start",
  "build": "react-app-rewired build",
  "test": "react-app-rewired test",
  "eject": "react-script eject"
},
....
```

3. 在根目录下创建config-overrides.js

```
const { override, fixBabelImports, addLessLoader } = require('customize-cra')
module.exports = override(
  fixBabelImports('import', {
    libraryName: 'antd',
    libraryDirectiry: 'es',
    style: true
  }),
  addLessLoader({
    lessOptions: {
      javascriptEnabled: true,
      modifyVars: {'@primary-color': 'green'}
    }
  })
)
```

4. 备注：不用在组件里亲自引入样式了，即：`import 'antd/dist/antd.css'`应该删掉
5. 移动端使用的组件库：vant

---

7. 第七章：redux

7.1. redux 理解

7.1.1. 学习文档

1. 英文文档: https://redux.js.org/
2. 中文文档: http://www.redux.org.cn/
3. Github: < https://github.com/reactjs/redux>

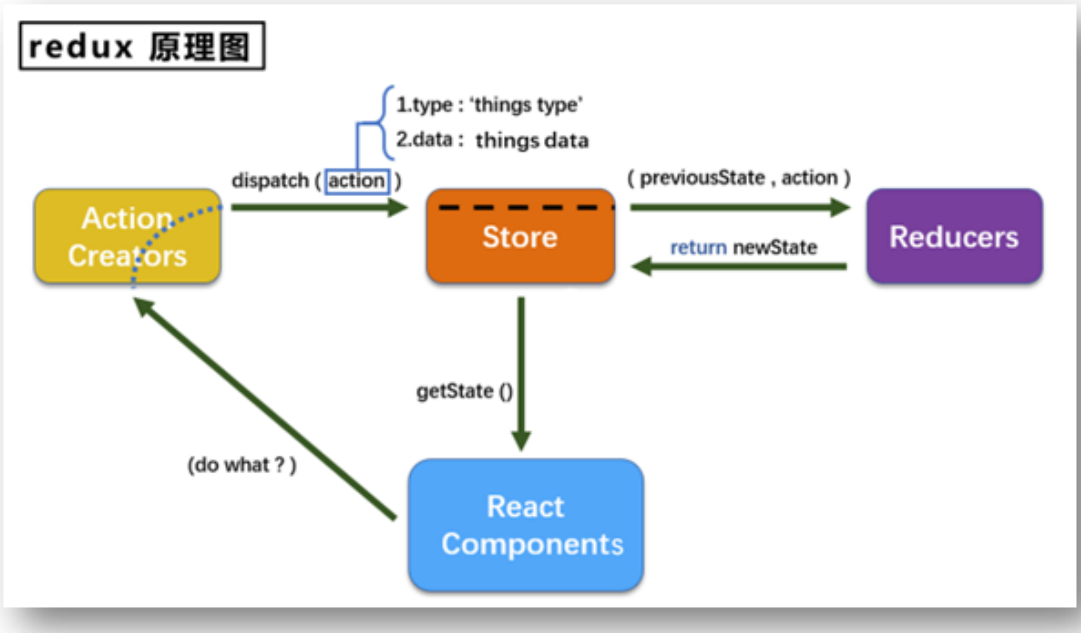
7.1.2. redux 是什么

1. redux 是一个专门用于做状态管理的 JS 库（不是 react 插件库）。
2. 它可以用在 react、angular、vue 等项目中，但基本与 react 配合使用。
3. 作用：集中式管理 react 应用中多个组件共享的状态。

7.1.3. 什么情况下需要使用 redux

1. 某个组建的状态，需要让其他组件随时可以拿到（共享）。
2. 一个组件需要改变另一个组件的状态（通信）。
3. 总体原则：能不用就不用，如果不用比较吃力才考虑使用。

7.1.4. redux 工作流程



7.2. redux 的三个核心概念

7.2.1. action

1. 动作对象
2. 包含 2 个属性
   - type：标识属性，值为字符串，唯一，必要属性
   - data：数据属性，值类型任意，可选属性
3. 例子：`{type: 'ADD_STUDENT', data: {name: 'tom', age: 18}}`
4. 代码示例：

```
/*
  该文件专门为Count组件生成action对象
*/
// 引入常量模块
import { INCREMENT, DECREMENT } from './constant'

export const createIncrementAction = (data) => ({ type: INCREMENT, data
})
export const createDecrementAction = (data) => ({ type: DECREMENT, data
})

```

7.2.2. reducer

1. 用于初始化状态、加工状态。
2. 加工时，根据旧的 state 和 action，产生新的 state 的 纯函数。
3. 代码示例

```
/*
  1.该文件是用于创建一个为Count组件服务的reducer，reducer的本质就是一个函数
  2.reducer函数会接到两个参数，分别为：之前的状态（preState），动作对象（action）
*/
// 引入常量模块
import { INCREMENT, DECREMENT } from "./constant"
const initState = 0 // 初始化状态
export default function countReducer(preState = initState, action) {
  console.log(preState)
  // 从action对象中获取：type、data
  const { type, data } = action
  // 根据type决定如何加工数据
  switch (type) {
    case INCREMENT: // 如果是加
      return preState + data
    case DECREMENT: // 如果是减
      return preState - data
    default:
      return preState
  }
}

```

7.2.3. store

1. 将 state、action、reducer联系在一起的对象
2. 任何得到此对象?
    1. `import {createStore} from 'redux'`
    2. `import reducer from './reducers'`
    3. `const store = createStore(reducer)`
3. 此对象的功能?
    1. getState()：得到 state
    2. dispatch(action)：分发 action，触发 reducer 调用，产生新的 state
    3. subscribe(listener)：注册监听，当产生了新的 state 时，自动调用
4. 代码示例：

```
/*
   该文件专门用于暴露一个store对象，整个应用只有一个store对象
*/

// 引入createStore，专门用于创建redux最为核心的store对象
import { legacy_createStore } from 'redux'
// 引入为Count组件服务的reducer
import countReducer from './count_reducer'

// 暴露store
export default legacy_createStore(countReducer)

```

7.3. redux 的核心 API

7.3.1. createstore()

作用：创建包含指定 reducer 的 store 对象

7.3.2. store 对象

1. 作用：redux 库最核心的管理对象
2. 它内部维护着：
   - state
   - reducer
3. 核心方法：
   1. getState()
   2. dispatch(action)
   3. subscribe(listener)
4. 具体编码：
   1. `store.getState()`
   2. `store.dispatch({type: 'INCREMENT', data})`
   3. `store.subscribe(render)` （监听redux状态的变化，如果发生变化，就重新render，放在App.jsx）

7.3.3. applyMiddleware()

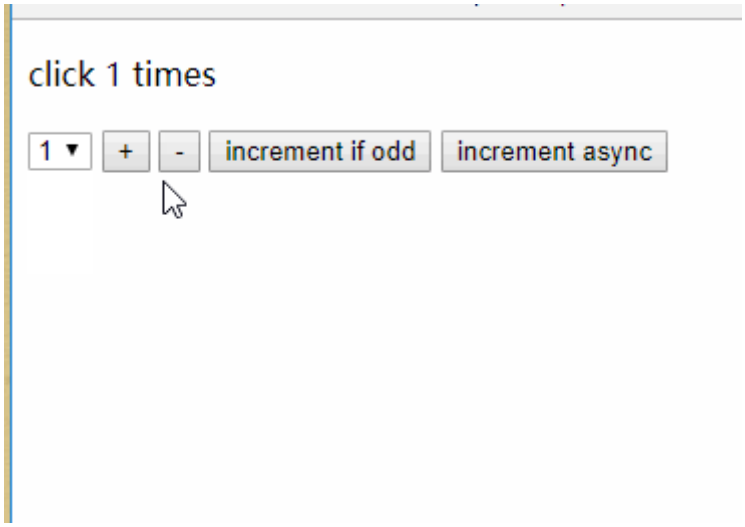
作用：应用上基于redux的中间件(插件库)

7.3.4. combineReducers()

作用：合并多个reducer函数

7.4. 使用 redux 编写应用

7.4.1. 效果



7.4.2. 求和案例 _redux 精简版

1. 去除 Count 组件自身的状态
2. src 下建立：
    - src
        - redux
            - store.js
            - count_reducer.js
3. store.js：
    1. 引入 redux 中的 createStore 函数，创建一个 store（createStore 将要被弃用了，建议使用 legacy_createStore 代替）
    2. createStore 调用时要传入一个为其服务的 reducer
    3. 记得暴露 store 对象
4. count_reducer.js
    1. reducer 的本质是一个函数，接收：preState, action，返回加工后的状态
    2. reducer 有两个作用：初始化状态，加工状态
    3. reducer 被第一次调用时，是 store 自动触发的，

        传递的 preState 是 undefined

        传递的action 是：{type:'@@REDUX/INIT_a.2.b.3}
5. 在 index.js 中监测 store 中状态的改变，一旦发生改变重新渲染 <App />

    备注：redux 只负责状态管理，至于状态的改变驱动着页面的显示，要靠我们自己写。

```
import React from 'react'
import ReactDOM from 'react-dom/client'
import App from './App'
import store from './redux/store'

const root = ReactDOM.createRoot(document.getElementById('root'))
root.render(
  <React.StrictMode>
    <App />
  </React.StrictMode>
)
store.subscribe(() => {
```

```
  root.render(
    <React.StrictMode>
      <App />
    </React.StrictMode>
  )
})

```

7.4.3. 求和案例 _redux 完整版

新增文件：

1. count_action.js：专门用于创建 action 对象
2. constant.js：放置容易写错的 type 值

7.4.4. 求和案例 _redux 异步action版

1. 明确：延迟的动作不想交给组件自身，想交给action
2. 何时需要异步action：想要对状态进行操作，但是具体的数据靠异步任务返回。
3. 具体编码：
    1. 下载中间件：`yarn add redux-thunk`，并配置在 store 中

```
/*
  该文件专门用于暴露一个store对象，整个应用只有一个store对象
*/

// 引入createStore，专门用于创建redux最为核心的store对象
// applyMiddleWare 执行中间件
import { legacy_createStore, applyMiddleWare } from 'redux'
// 引入为Count组件服务的reducer
import countReducer from './count_reducer'
// 引入redux-thunk
import thunk from 'redux-thunk'

// 暴露store
export default legacy_createStore(countReducer,
applyMiddleWare(thunk))

```

    2. 创建 action 的函数不是返回一般对象（同步action），而是返回一个函数，该函数中写异步任务。

```
// 异步action，就是指action的值为函数，异步action中一般都会调用同步action，
异步action不是必须要用的
export const createIncrementAction = (data) => ({ type: INCREMENT,
data })
export const createDecrementAction = (data) => ({ type: DECREMENT,
data })
export const createIncrementAsyncAction = (data, time) => {
  return (dispatch) => {
    setTimeout(() => {
      dispatch(createIncrementAction(data))
    }, time)
  }
}
```

3. 异步任务有结果后，分发一个同步的action去真正操作数据。

4. 备注：异步 action 不是必须要写，完全可以让组件自己等待异步任务的结果出来了再去分发同步 action。

7.4.5. 求和案例 _react-redux 基本使用

书写步骤：

1. 在之前书写好的redux基础上，删掉 UI 组件上的所有跟 redux 相关的 东西
2. 书写容器组件

```
/* 容器组件 */
// 引入Count的UI组件
import CountUI from '../../components/Count'
// 引入action
import { createIncrementAction, createDecrementAction,
createIncrementAsyncAction } from '../../redux/count_action'

// 引入connect用于链接UI组件与redux
import { connect } from 'react-redux'

// 使用connect()()创建并暴露一个Count的容器组件
export default connect(
  // mapStateToProps 函数返回的对象中的key就作为传递给UI组件props的key，value就作为
传递给UI组件props的value —— 状态
  state => ({ count: state }),

  // mapDispatchToProps的一般写法
  // mapDispatchToProps 函数返回的对象中的key就作为传递给UI组件props的key，value就
作为传递给UI组件props的value —— 操作状态的方法
  /* dispatch => ({
    jia: number => dispatch(createIncrementAction(number)),
    jian: number => dispatch(createDecrementAction(number)),
    jiaAsync: (number, time) => dispatch(createIncrementAsyncAction(number,
time))
  }) */

  // mapDispatchToProps的简写
  {
    jia: createIncrementAction,
    jian: createDecrementAction,
    jiaAsync: createIncrementAsyncAction
  }
)(CountUI)

```

1. 明确两个概念：
    1. UI 组件：不能使用任何 redux 的 API，只负责页面的呈现、交互等。
    2. 容器组件：负责和 redex 通信，将结果交给 UI 组件。
2. 如何创建一个容器组件——靠 react-redux 的 connect 函数
    - `connect(mapStateToProps, mapDispatchToProps)(UI组件)` —— map 有映射的意思
        - mapStateToProps(函数)：映射状态，返回的是一个对象
        - mapDispatchToProps(函数)：映射操作状态的方法，返回值是一个对象
3. 备注1：容器组件中的 store 是在App组件中靠 props 传进去的，而不是在容器组件中直接引入

```jsx
// App.jsx
import Count from './container/Count'
import store from './redux/store'

<Count store={store}/>
```

4. 备注2：mapDispatchToProps 可以传两种值，可以是函数（返回一个对象），也可以是一个对象（只需要准备好action，react-redux 会自动分发）

```
  // mapDispatchToProps的简写
  {
    jia: createIncrementAction,
    jian: createDecrementAction,
    jiaAsync: createIncrementAsyncAction
  }
```

7.4.6. 求和案例 _react-redux 优化

1. 容器组件和 UI 组件整合成一个文件
2. 无需自己给容器组件传递 store，给 <App/> 包裹一个 `<Provider store={store}>` 即可

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

3. 使用了 react-redux 后也不用再自己检测 redux 中状态的改变了，容器组件可以自动完成这个工作（因为是通过 `connect()()` 创建的。
4. mapDispatchToProps 也可以简单的写成一个对象
5. 一个组件要和 redux “打交道” 要经过哪几步？
   1. 定义好 UI 组件 —— 不暴露
   2. 从 react-redux 引入 connect 生成一个容器组件，并暴露，写法如下：

```
connect(
  state => ({key: value}), // 映射状态
  {key: xxxxAction} // 映射操作状态的方法
)(UI组件)
```

   3. 在 UI 组件中通过this.props.xxxxx读取和操作状态

7.4.7. 求和案例 _react-redux 数据共享版

1. 定义一个 Person 组件，和 Count 组件通过 redux 共享数据。
2. 为 Person 组件编写：reducer、action，配置 constant 常量。
3. 重点：Person 组件的 reducer 和 Count 组建的 reducer 要使用 combineReducers 进行合并，合并后的总状态是一个对象！！！

```
/*
  redux/index.js
  该文件用于汇总所有的reducer为一个总的reducer
*/
// 引入combineReducers,用于汇总多个reducer
```

```
import { combineReducers } from 'redux'
// 引入为Count组件服务的reducer
import count from './count'
// 引入为Person组件服务的reducer
import persons from './person'

// 汇总并暴露所有的reducer变为一个总的reducer
export default combineReducers({
  count,
  persons,
})



//store.js
/*
  该文件专门用于暴露一个store对象，整个应用只有一个store对象
*/
// 引入 legacy_createStore，专门用于创建redux最为核心的store对象
import { legacy_createStore, applyMiddleware } from 'redux'
// 引入汇总之后的reducer
import reducer from './reducers'
// 引入redux-thunk，用于支持异步action
import thunk from 'redux-thunk'
// 引入redux-devtools-extension，用于支持开发者工具
import { composeWithDevTools } from 'redux-devtools-extension'

// 创建并暴露store
export default legacy_createStore(reducer,
composeWithDevTools(applyMiddleware(thunk)))
```

4. 交给 store 的是总 reducer，最后注意在组件中取出状态的时候，记得 "取到位"。

7.4.8. 求和案例 _react-redux 最终版

1. 所有变量名字要规范，尽量触发对象的简写形式。
2. reducers 文件夹中，编写 index.js 专门用于汇总并暴露所有的 reducer。

7.5. redux 异步编程

7.5.1. 理解:

1. redux 默认是不能进行异步编程的
2. 某些时候应用中需要在 redux 中执行异步任务(ajax，定时器)

7.5.2. 使用异步中间件

1. 下载

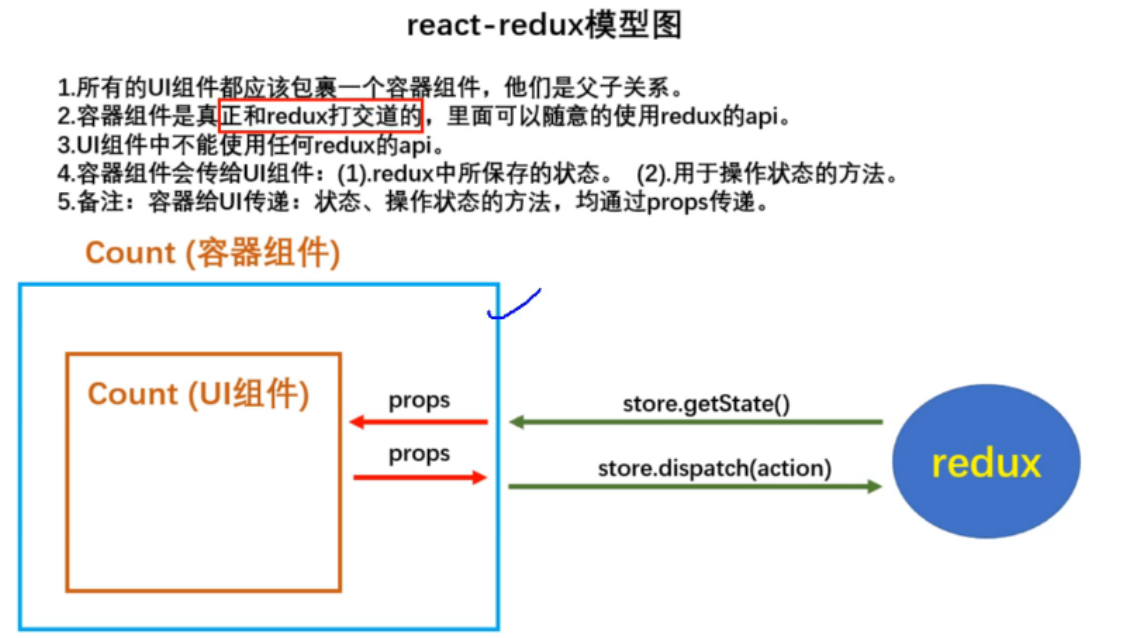
```
npm install --save redux-thunk
```

2. 使用

```
//store.js
/*
  该文件专门用于暴露一个store对象，整个应用只有一个store对象
*/
```

```
// 引入 legacy_createStore，专门用于创建redux最为核心的store对象
import { legacy_createStore, applyMiddleware } from 'redux'
// 引入汇总之后的reducer
import reducer from './reducers'
// 引入redux-thunk，用于支持异步action
import thunk from 'redux-thunk'
// 引入redux-devtools-extension，用于支持开发者工具
import { composeWithDevTools } from 'redux-devtools-extension'

// 创建并暴露store
export default legacy_createStore(reducer,
composeWithDevTools(applyMiddleware(thunk)))
```

7.6. react-redux



1. 都有的 UI 组件库都应该包裹一个容器组件，他们是父子关系。
2. 容器组件是真正和 redux 打交道的，里面可以随意的使用 redux 的 API 。
3. UI 组件中不能使用任何 redux 的 API。
4. 容器组件会传给 UI 组件：
    1. redux 中所保存的状态。
    2. 用于操作状态的方法。
5. 备注：容器给 UI 组件传递：redux 里的状态、操作 redux 里的状态方法，均通过 props 传递。

7.6.1. 理解

1. 一个 react 插件库
2. 专门用来简化 react 应用中使用 redux

7.6.2. react-redux 将所有组件分成两大类

1. UI 组件
    1. 只负责 UI 的呈现，不带有任何业务逻辑
    2. 通过 props 接收数据（一般数据和函数）
    3. 不使用任何 Redux 的API
    4. 一般保存在 components 文件夹下
2. 容器组件
    1. 负责管理数据和业务逻辑，不负责 UI 的呈现
    2. 使用 Redux 的 API
    3. 一般保存在 containers 文件夹下

7.6.3. 相关 API

1. Provider：让App中所有需要使用 store 的组件都可以得到 store

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

// 在App中不需要再给组件传递store，Provider会自动传递
```

2. connect：用于包装 UI 组件生成容器组件

```
import { connect } from 'react-redux'

connect(
  mapStateToProps,
  mapDispatchToProps
)(Counter)
```

3. mapStateToprops：将外部的数据（即 state 对象）转换为 UI 组件的标签属性（props
    1. mapStateToProps 函数的返回的是一个对象；
    2. 返回的对象中的key就作为传递给UI组件props的key，value就作为传递给UI组件props的value

    3.mapStateToProps 用于传递状态

```
const mapStateToProps = function(state) {
  return {
    value: state
  }
}
```

4. mapDispatchToProps：将分发 action 的函数转换为 UI 组件的标签属性

    mapDispatchToProps 函数的返回的对象中的key就作为传递给UI组件props的key，value就作为传递给UI组件props的value————操作状态的方法

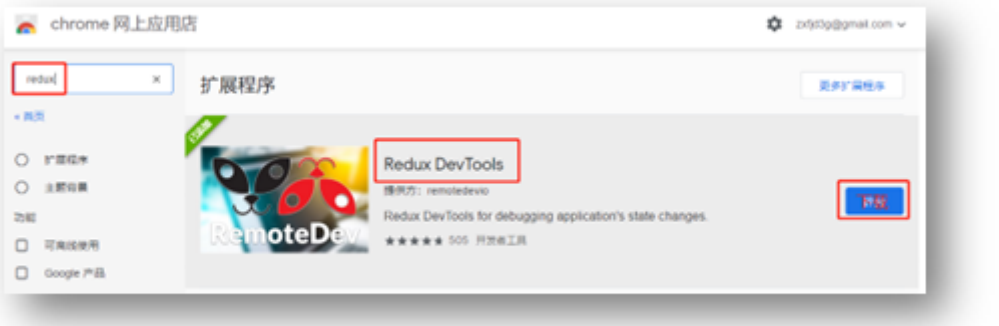
```
  // mapDispatchToProps的一般写法
  /* dispatch => ({
    jia: number => dispatch(createIncrementAction(number)),
    jian: number => dispatch(createDecrementAction(number)),
    jiaAsync: (number, time) =>
dispatch(createIncrementAsyncAction(number, time))
  }) */

  // mapDispatchToProps的简写
  {
    jia: createIncrementAction,
    jian: createDecrementAction,
    jiaAsync: createIncrementAsyncAction
  }
```

- mapDispatchToProps 可以传两种值：
  1. function
  2. {}

     只需要返回一个 action，react 会自动帮你分发（dispatch）

7.7. 使用 redux 调试工具

7.7.1. 安装 chrome 浏览器插件



7.7.2. 下载工具依赖包

1. 下载插件库：

```
yarn add redux-devtools-extension
```

2. 在 store 中引入，并配置

```
import { composeWithDevTools } from 'redux-devtools-extension'

// export default createStore(allReducer, applyMiddleware(thunk))  // 不用redux调试工具的写法
export default createStore(allReducer, composeWithDevTools(applyMiddleware(thunk)))

```

7.8. 纯函数和高阶函数

7.8.1. 纯函数

1. 一类特别的函数：只要是同样的输入（实参），必定得到同样的输出（返回）
2. 必须遵守以下一些约束
    1. 不得改写参数数据
    2. 不会产生任何副作用，例如网络请求、输入和输出设备
    3. 不能调用 Date.now() 或者 Math.random() 等不纯的方法
3. redux 的 reducer 函数必须是一个纯函数

7.8.2. 高阶函数

1. 理解：一类特别的函数
    1. 情况1：参数是函数
    2. 情况2：返回是函数
2. 常见的高阶函数：
    1. 定时器设置函数
    2. 数组的 forEach() / map() / filter() / reduce() / find() / bind()
    3. promise
    4. react-redux 中的 connect 函数
3. 作用：能实现更新动态，更新扩展的功能

---

8. 第八章：React 扩展

8.1. setState

React 状态的更新是异步的，setState是同步的，但是 setState 引起的后续更新动作是异步的。

8.1.1. setState 更新状态的 2 种写法

1. setState(stateChange, [callback]) ------对象式的setState
    1. stateChange 为状态改变对象（该对象可以体现出状态的更改）
    2. callback 是可选的函数，它在状态更新完毕、界面也更新后（render 调用后）才被调用
2. setState(updater, [callback]) ----- 函数式的setState
    1. updater 为返回 stateChange 对象的函数。
    2. updater 可以接收到 state 和 props。
    3. callback 是可选函数，它在状态更新、界面也更新后（render 调用后）才被调用。

总结：

1. 对象式的 setState 是函数式 setState 的简写方式（语法糖）
2. 使用原则：
    1. 如果新状态不依赖于原状态 ===> 使用对象方式
    2. 如果新状态依赖于原状态 ===> 使用函数方式
    3. 如果需要在 setState() 执行后获取最新的状态数据，要在第二个参数 callback 函数中读取

8.2. lazyLoad

8.2.1. 路由组件的 lazyLoad

```
import { lazy } from 'react'
import Loading from './Loading'

// 1.通过 React 的 lazy 函数配合 import() 函数动态加载路由组件 ===> 路由组件代码会被分开打包
const Login = lazy(() => import('@/pages/Login'))

// 2.通过 <Suspense> 指定在加载器得到路由打包文件前显示一个自定义的loading 界面
<Suspense fallback={<Loading />}>  // 这里可以写好一个加载中的组件
  <Switch>
    <Route path="/xxx" component={Xxxx} />
    <Redirect to="/login" />
  </Switch>
</Suspense>
```

8.3. Hooks

8.3.1. React Hook / Hooks 是什么?

1. Hook 是 React 16.8.0 版本增加的新特性 / 新语法
2. 可以让你在函数组件中使用 state 以及其他的 React 特性

8.3.2. 三个常用的 Hook

1. State Hook：React.useState()
2. Effect Hook：React.useEffect()
3. Ref Hook：React.useRef()

8.3.3. State Hook

1. State Hook 让函数组件也可以有 state 状态，并进行状态数据的读写操作
2. 语法：

```
// import React,{useState} from 'react' // 也可以不用React.直接这里引入，下面直接写useState()

const [xxx, setXxx] = React.useState(initValue)
```

3. useState() 说明：
   - 参数：第一次初始化指定的值在内部作缓存
   - 返回值：包含 2 个元素的数组，第 1 个为内部当前状态值，第二个为更新状态值的函数
4. setXxx() 2 种写法：
   - setXxx(newValue)：参数为非函数值，直接指定新的状态值，内部用其覆盖原来的状态值
   - setXxx(value => newValue)：参数为函数，接收原本的状态值，返回新的状态值，内部用其覆盖原来的状态值

8.3.4. Effect Hook

1. Effect Hook 可以让你在函数组件中执行副作用操作（用于模拟类组件中的生命周期钩子）
2. React 中的副作用操作：
   - 发 ajax 请求数据获取
   - 设置订阅 、启动定时器
   - 手动更新真实 DOM
3. 语法和说明：

```
useEffect(() => {  // 第一个参数为函数，第二个参数是一个数组
      // 在此可以执行任何带副作用操作
      return () => { // 在组件卸载前执行
      // 在此做一些收尾工作，比如清除定时器/取消订阅等
  }
}, [stateValue]) // 如果指定的是[]，回调函数只会在第一次render() 后执行，这里传入的 [] 表示监测的作用，[]表示什么都不监测，不传表示全部监测，[count]表示监测count状态
```

4. 可以把 useEffect Hook 看做如下三个函数的组合
   - componentDidMount()：第二个参数传入的是`[]`第一个参数（函数）中就相当于componentDidMount钩子
   - componentDidUpdate()：在第一个参数函数中，第二个参数不传
   - componentWillUnmount()：第一个函数参数里面返回的函数内部就相当于componentWillUnmount

8.3.5. Ref Hook

1. Ref Hook 可以在函数组件中存储 / 查找组件内的标签或任意其它数据
2. 语法：`const refContainer = useRef()` ==>构建一个`ref`容器，保存标签对象
3. 作用：保存标签对象，功能与 React.createRef() 一样

8.4. Fragment

8.4.1. 使用

```

<Fragment>  // 原本的最外层包裹的div换成 fragment，需要先从 React 引入
  <h1></h1>
  <button></button>
</Fragment>  // 能传一个 key属性

<></>  // 什么属性都不能传
```

8.4.2. 作用

去掉不必要的 div 包裹，不再把 div 渲染成真实 DOM 了

可以不用必须有一个真实的 DOM 根标签

8.5. Context

8.5.1. 理解

一种组件间通信方式，常用于【祖组件】与【后代组件】间通信。生产者——消费者模式

8.5.2. 使用

1. 创建 Context 容器对象

```
// 创建contex对象(组件对象，首字母大写)
const XxxContext = React.createContext()
```

2. 渲染子组件时，外面包裹 xxxContext.Provider，通过 value 属性给后代组件传递数据，value需要传递多个数据的时候写成对象，取出的时候记得“取到位”：

```
<XxxContext.Provider value={数据}> // 这里属性名必须叫value
  子组件
</XxxContext.Provider>
```

3. 后代组件读取数据：

```
// const { Provider, Consumer } = XxxContext
// 第一种方式：仅适用于类组件
static contextType = XxxContext // 声明接收context
this.context // 读取 context 中的value数据

// 第二种方式：函数组件与类组件都可以
<XxxContext.Consumer>
  {
    value => {
      要显示的内容
    }
  }
</xxxContext.Consumer>
```

```
// 函数式组件接收context参数
function C() {
  return (
    <div className="grand">
      <h3>我是C组件</h3>
      <h4>
        我从A组件收到的用户名是:
        <Consumer>
          {value => `${value.username}, 我的年龄是:${value.age}`}
        </Consumer>
      </h4>
    </div>
  )
}
```

8.5.3. 注意

在应用开发中一般不用context，一般都用它封装 react 插件

8.6. 组件优化

8.6.1. Component 的 2 个问题

1. 只要执行 setState()，即使不改变状态数据，组件也会重新 render() ===> 效率低
2. 只要当前组件重新 render()，就会重新 render 子组件，纵使子组件没有用到父组件的任何数据 ===> 效率低

8.6.2. 效率高的做法

只有当组件的 state 或 props 数据发生改变时才重新 render()

8.6.3. 原因

Component 中的 shouldComponentUpdate() 总是返回 true

8.6.4. 解决

1. 办法1：
    - 重写 shouldComponentUpdate() 方法
    - 比较新的 state 或 props 数据，如果有变化才返回 true，如果没有返回 false

```
import React, { Component } from 'react'
import './index.css'

export default class Parent extends Component {
  state = { carName: '红旗HS5' }

  changeCar = () => {
    this.setState({ carName: '雷克萨斯' })
  }

  shouldComponentUpdate(_, nextState) {
    // 当参数使用不到需要占位时使用 _
    // console.log(this.state, this.props) // 目前的props和state
    // console.log(nextProps, nextState) // 接下来要变化的目标props，目标state
    return !(this.state.carName === nextState.carName)
    // return true
  }

  render() {
    console.log('Parent---render')
    const { carName } = this.state
    return (
      <div className="parent">
        <h3>我是Parent组件</h3>
        <span>我的车名字是：{carName}</span>
        <br />
        <button onClick={this.changeCar}>点我换车</button>
        <Child carName={carName} />
```

```
        </div>
      )
    }
}

class Child extends Component {
  shouldComponentUpdate(nextProps, _) {
    // 当参数使用不到需要占位时使用 _
    // console.log(this.state, this.props) // 目前的props和state
    // console.log(nextProps, nextState) // 接下来要变化的目标props，目标state
    return !(this.props.carName === nextProps.carName)
  }

  render() {
    console.log('Child---render')
    return (
      <div className="child">
        <h3>我是Child组件</h3>
        <span>我接到的车是: {this.props.carName}</span>
      </div>
    )
  }
}
```

2. 办法2：
   - 使用 PureComponent 代替 Component
   - PureComponent 重写了 shouldComponentUpdate()，只有 state 或 props 数据有变化才返回 true
   - 注意：
     - 只是进行 state 和 props 数据的浅比较，如果数据对象内部的数据变了，返回 false
     - 不要直接修改 state 数据，而是要产生新数据
     - 当你进行 setState 的时候，一定不要跟原来的状态对象发生任何关联，否则不能更新

项目中一般使用 PureComponent 来优化

8.7. render props

8.7.1. 如何向组件内部动态传入带内容的结构（标签）？

```
Vue中：
  使用 slot(插槽) 技术，也就是通过组件标签体传入结构 <A><B /></A>
React中：
  使用children props：通过组件标签体传入结构
  使用 render props：通过自建标签属性传入结构，而且可以携带数据，一般用 render 函数属性
```

8.7.2. children props

```
<A>
  <B>xxxx</B>
</A>
{this.props.children}
问题：如果 B 组件需要 A 组件内的数据， ===> 做不到
```

8.7.3. render props

```
<A render={(data) => <C data={data}></C>}></A>  // A组件书写的地方（A的父组件里）
A 组件内部需要放置的地方：{this.props.render(内部 state 数据)}
C 组件：读取 A 组件传入的数据显示 {this.props.data}
```

代码示例：

```
import React, { Component } from 'react'
import './index.css'

export default class Parent extends Component {
  render() {
    return (
      <div className="parent">
        <h3>我是Parent组件</h3>
        <A render={(name) => <B name={name} />} />
      </div>
    )
  }
}

class A extends Component {
  state = { name: 'tom' }
  render() {
    // console.log(this.props)
    const { name } = this.state
    return (
      <div className="a">
        <h3>我是A组件</h3>
        {this.props.render(name)}
      </div>
    )
  }
}
class B extends Component {
  render() {
  console.log(this.props.name)
    return (
      <div className="b">
        <h3>我是B组件</h3>
      </div>
    )
  }
}
```

8.8. 错误边界

8.8.1. 理解：

错误边界（Error boundary）：用来捕获后代组件错误，渲染备用页面

如果该组件的子组件出现了任何的报错，都会调用`getDerivedStateFromError`，需要返回一个错误状态对象

8.8.2. 特点：

只能捕获后代组件生命周期产生的错误，不能捕获自己组件产生的错误和其他组件在合成事件、定时器中产生错误

8.8.2.1. 使用方式：

`getDerivedStateFromError` 配合 `componentDidCatch()`

```
state = { hasError: ''}

// 生命周期函数，一旦后台组件报错，就会触发
static getDerivedStateFromError(error) {
  console.log(error)
  // 在 render 之前触发
  // 返回新的 state
  return {
    hasError: error
  }
}

// 渲染组件时，如果你的子组件出现错误，就会调用componentDidCatch
// componentDidCatch 钩子一般用来统计错误次数，反馈给服务器，用于通知编码人员进行bug的解决
componentDidCatch(error, info) {
  // 统计页面的错误。把请求发送到后台去，用于通知编码人员进行bug的解决
  condole.log(error,info)
}
```

8.9. 组件通信方式总结

8.9.1. 组件间的关系

- 父子组件
- 兄弟组件（非嵌套组件）
- 祖孙组件（跨级组件）

8.9.2. 几种通信方式：

1. props：
    - children props
    - render props
2. 消息订阅——发布：

- pubs-sub、event等等

3. 集中式管理：
   - redux、dva等等
4. conText：
   - 生产者——消费者模式

8.9.3. 比较好的搭配方式：

- 父子组件：props
- 兄弟组件：消息订阅——发布、集中式管理（redux）
- 祖孙组件（跨级组件）：消息订阅——发布、集中式管理、conText（开发时用的少，封装插件用的多）

---

9. 第九章：React Router 6

9.1. 概述

1. React Router 以三个不同的包发布到 npm 上，它们分别为：
   1. react-router：路由的核心库，提供了二环内多的：组件、钩子。
   2. **react-router-dom：包含 react-router 所有内容，并添加了一些 DOM 的组件，例如 `<BrowserRouter>` 等。**
   3. react-router-native：包括 react-router 的所有内容，并添加了一些专门用于 ReactNative 的 API，例如：`<NativeRouter>`（重定向）等。
2. 与 React Router 5.x 版本相比，改变了什么?
   1. 内置组建的变化：移除了`<Switch/>`，新增了`<Routers/>`等。
   2. 语法的变化：注册路由时`<component={About}`变为`element={<About/>}`等。
   3. 新增多个 hook：`useParams`、`useNavigate`、`useMatch`等。
   4. 官方明确推荐函数式组件了！！！

9.2. Componnet

9.2.1. `<BrowserRouter>`

1. 说明：`<BrowserRouter>`用于包裹整个应用（App组件）
2. 示例代码：

```
import React from 'react'
import ReactDOM from 'react-dom/client'
import { BrowserRouter } from 'react-router-dom'
import App from './App'

const root = ReactDOM.createRoot(document.getElementById('root'))
root.render(
  <React.StrictMode>
    <BrowserRouter>
      {/* 整体结构通常为App组件 */}
      <App />
```

```
    </BrowserRouter>
  </React.StrictMode>
)
```

9.2.2. `<HashRouter>`

1. 说明：作用与`<BrowserRouter>`一样，但`<HashRouter>`修改的式地址栏的hash值
2. 备注：6.x 版本中`<HashRouter>`、`<BrowserRouter>`的用法与 5.x 版本相同。

9.2.3. `<Routes />` 和 `<Route>`

1. v6 版本移除了先前的`<Switch>`，引入了新的替代者：`<Routes>`。
2. `<Routes>`和 `<Route />`要配合使用，且必须要用`<Routes>`包裹`<Route />`。
3. `<Route>`相当于一个 if 语句，如果其路径与当前 URL 匹配，则呈现对应的组件。
4. `<Route caseSenaitive>`属性用于指定：匹配时是否区分大小写（默认 false）。
5. 当 URL 发生变化时，`<Routes>`都会查看其所有的子`<Route />`以找到最佳匹配并呈现组件。
6. `<Route/>`也可以嵌套使用，且可配合`useRoutes()`配饰"路由表"，但需要通过`<Outlet>`组件来指定其子路由渲染的地方。
7. 示例代码：

```
<Routes>
  {/* path属性用于定义路径，element属性用于定义当前路径所对应的组件 */}
  <Route path="/about" element={<About />} />
  {/* 用于定义嵌套路由，home是一级路由，对应的路径/home */}
  <Route path="/home" element={<Home />} >
    <Route path="/test" element={<Test />} ></Route>
    <Route path="/test1" element={<Test1 />} ></Route>
  <Route/>

  {/* Route也可以不写element属性，这时就是用于展示嵌套的路由，所对应的路径是/users/xxx */}
  <Route path="users" >
    <Route path="/xxx" element={<Demo />} />
  </Route>
</Routes>
```

9.2.4. `<Link>`

1. 作用：修改 URL，且不发送网络请求（路由链接）。
2. 注意：外侧需要用`<BrowserRouter>`或`<HashRouter>`包裹。
3. 示例代码：

```
import { Link } from 'react-router-dom'

function Test() {
  return (
    <div>
      <Link to="路径">按钮</Link>
    </div>
  )
}
```

9.2.5. `<NavLink>`

1. 作用：与`Link`组件类似，且可实现导航的 “高亮” 效果。
2. 当NavLink上添加了 end 属性后，若 Home 的子组件匹配成功，则Home的导航没有高亮效果。
3. 示例代码：

```
// 注意：NavLink 默认类名是 active，下面指定自定义的类名
// isActive NavLink 是否被点击高亮，是加什么类名，不是又加什么类名

// 自定义样式
<NavLink
  to="login"
  className={({isActive}) => {
    console.log('home', isActive)
    return isActive ? 'base one' : 'base' // one属性是要修改成的样式，base是基础高亮的样式
  }}
>login</NavLink>

/*
  默认情况下，当Home的子组件匹配成功，Home的导航就会高亮，
  当NavLink上添加了 end 属性后，若 Home 的子组件匹配成功，则Home的导航没有高亮效果。
*/
<NavLink to="home" end >home</NavLink>
```

9.2.6. `<Navigate>`

1. 作用：只要`<Navigate>`组件被渲染，就会修改路径，切换视图。也代替之前的 Redirect 使用
2. `replace`属性用于控制跳转模式（push 或 replace，默认是 push）。
3. 示例代码：

```
{/* 创建路由链接 */}
<NavLink className="list-group-item" to="/about">
  About
</NavLink>
<NavLink className="list-group-item" to="/home">
  Home
</NavLink>

{/* 注册路由 */}
<Routes>
  <Route path="/home" element={<Home />} />
  <Route path="/" element={<Navigate to="/home" />} />
</Routes>
```

```
import React, { useState } from 'react'
import { Navigate } from 'react-router-dom'

export default function Home() {
  const [sum, setSum] = useState(1)
  return (
```

```
    <div>
      <h3>我是Home的内容</h3>
      {/* 根据sum的值决定是否切换视图 */}
      {sum === 2 ? <Navigate to="/about" replace={true} /> : <h4>当前sum的值为: {sum}</h4>}
      <button onClick={() => setSum(2)}>点我将sum变为2</button>
    </div>
  )
}

```

9.2.7. `<Outlet>`

1. 当Route产生嵌套式，渲染其对应的后续子路由。
2. 示例代码：

```
// 根据路由表生成对应的路由规则
import { useRoutes } from 'react-router-dom'

const element = useRoutes([
  {
    path: '/about',
    element: <About />
  },
  {
    path: '/home',
    element: <Home />,
    children: [
      {
        path: 'message',
        element: <Message />
      },
      {
        path: 'news',
        element: <News />
      }
    ]
  },
  {
    path: '/',
    element: <Navigate to="/about" />
  }
])

// Home.js
import React from 'react'
import { NavLink, Outlet } from 'react-router-dom'

export default function Home() {
  return (
    <div>
      <ul className="nav nav-tabs">
        <li>
          <NavLink className="list-group-item" to="news">
            News
```

```
          </NavLink>
        </li>
        <li>
          <NavLink className="list-group-item " to="message">
            Message
          </NavLink>
        </li>
      </ul>
      {/* 指定路由组件呈现的位置 */}
      <Outlet />
    </div>
  )
}

```

9.3. Hooks

9.3.1. useRoutes()

1. 作用：根据路由表，动态创建`<Routes>`和`<Route>`
2. 示例代码：

```
// 路由表配置：src/routes/index.jsx
import { Navigate } from 'react-router-dom'
import About from '../pages/About'
import Home from '../pages/Home'
import Message from '../pages/Message'
import News from '../pages/News'
import Detail from '../pages/Detail'

export default [
  {
    path: '/about',
    element: <About />
  },
  {
    path: '/home',
    element: <Home />,
    children: [
      {
        path: 'message',
        element: <Message />,
        children: [
          {
            path: 'detail',
            element: <Detail />
          }
        ]
      },
      {
        path: 'news',
        element: <News />
      },
      {

```

```
        path: './',
        element: <Navigate to="news" />
      }
    ]
  },
  {
    path: '/',
    element: <Navigate to="/about" />
  }
]

// App.jsx
import React from 'react'
import { NavLink, useRoutes } from 'react-router-dom'
import routers from './routers'
import Header from './components/Header'

export default function App() {
  const element = useRoutes(routers)
  return (
    <div classNameName="list-group">
        {/* 路由链接 */}
        <NavLink classNameName="list-group-item" to="/about">
            About
        </NavLink>
        <NavLink classNameName="list-group-item" end to="/home">
              Home
        </NavLink>
    </div>
    <div classNameName="col-xs-6">
      <div classNameName="panel">
        <div classNameName="panel-body">
          {/* 注册路由 */}
          {element}
        </div>
      </div>
    </div>
  )
}

```

9.3.2. useNavigate()

1. 作用：返回一个函数用来实现编程式路由导航。
2. 示例代码：

```
import React from 'react'
import { useNavigate } from 'react-router-dom'

export default function Header() {
  const navigate = useNavigate()
  const handle = () => {
    // 第一种使用方式：指定具体路径
    navigate('/login', {
      replace: false,
```

```
      state: {
        a: 1,
        b: 2
      }
    })

    // 第二种使用方式：传入数值进行前进或者后退，类似于 5.x 中的 history.go() 方法
    function skip() {
      navigate(-1) // 后退一步
      navigate(1) // 前进一步
    }
  }

  return (
    <div>
      <button onClick={handle}>按钮</button>
    </div>
  )
}
```

9.3.3. useParams()

1. 作用：返回当前匹配路由的`params`参数，类似于 5.x 中的 `match.params`。
2. 示例代码：

```
import React from 'react'
import { Routes, Route, useParams } from 'react-router-dom'
import User from './pages/User'

function ProfilePag() {
  // 获取URL 中携带过来的params参数
  let { id } = useParams()
}

function App() {
  return (
    <Routes>
      <Route path="users/:id" element={<User />} />
    </Routes>
  )
}
```

9.3.4. useSearchParams()

1. 作用：用于读取和修改当前位置的URL 中的查询字符串。
2. 返回一个包含两个值得数组，内容分别为：当前的 search参数、更新search的函数。
3. 示例代码：

```
import React from 'react'
import { useSearchParams, useLocation } from 'react-router-dom'

export default function Detail() {
  const [search, setSearch] = useSearchParams()
  const id = search.get('id')
```

```
  const title = search.get('title')
  const content = search.get('content')
  const x = useLocation()
  console.log(x)
  return (
    <ul>
      <li>
        <button onClick={() => setSearch('id=005&title=很好&content=嘻嘻')}>点我更新以下收到的search参数</button>
      </li>
      <li>消息编号: {id}</li>
      <li>消息标题: {title}</li>
      <li>消息内容: {content}</li>
    </ul>
  )
}
```

9.3.5. useLocation()

1. 作用：获取当前 location 信息，对标 5.x 中的路由组件的 `location` 属性。可以用来获取路由组件携带过来的`state`参数
2. 示例代码：

```
import React from 'react'
import { useLocation } from 'react-router-dom'

export default function Detail() {
  const {
    state: { id, title, content }
  } = useLocation()
  return (
    <ul>
      <li>消息编号: {id}</li>
      <li>消息标题: {title}</li>
      <li>消息内容: {content}</li>
    </ul>
  )
}
```

9.3.6. useMatch()

1. 作用：返回当前匹配信息，对标 5.x 中的路由组件的 `match` 属性。
2. 示例代码：

```
<Route path="/login/:page/:pageSize" element={<Login />} />
<NavLink to="/login/1/10">登录</NavLink>

export default function Login() {
  const match = useMatch('/login/:x/:y')
  console.log(match) // 输出match对象
  // match 对象内容如下：
  /*
    {
```

```
      params: {x: '1', y: '10'}
      pathname: 'Login/1/10'
      pathnameBase: 'Login/1/10'
      patten: {
        path : '/login/:x/:y',
        caseSensitive: false,
        end: false
      }
    }
  */
  return (
    <div>
      <h1>Login</h1>
    </div>
  )
}
```

9.3.7. UseInRouterContext()

作用：如果组件在`<Router>`的上下文中呈现，则`useInRouterContext`钩子返回 true，否则返回 false

9.3.8. useNavigationType()

1. 作用：返回当前的导航类型（用户是如何来到当前页面的）。
2. 返回值：`POP`、`PUSH`、`REPLACE`。
3. 备注：`POP`是指在浏览器中直接打开了这个路由组件（刷新页面）。

9.3.9. useOutlet()

1. 作用：用来呈现当前组件中要渲染的嵌套路由。
2. 示例代码：

```
const result = useOutlet()
console.log(result)

// 如果嵌套路由没有挂载，则result为null
// 如果嵌套路由已经挂载，则展示嵌套的路由对象
```

9.3.10. useResolvedPath()

1. 作用：给定一个 URL 值，解析其中的：path、search、hash值。