

React 全家桶(技术栈)

尚硅谷前端研究院

第 1 章：React 入门

1.1. React 的基本认识

1.1.1. 官网

- 1) 英文官网: <https://reactjs.org/>
- 2) 中文官网: <https://doc.react-china.org/>

1.1.2. 介绍描述

- 1) 用于构建用户界面的 JavaScript 库(只关注于 View)
- 2) 由 Facebook 开源

1.1.3. React 的特点

- 1) Declarative(声明式编码)
- 2) Component-Based(组件化编码)
- 3) Learn Once, Write Anywhere(支持客户端与服务器渲染)
- 4) 高效
- 5) 单向数据流

1.1.4. React 高效的原因

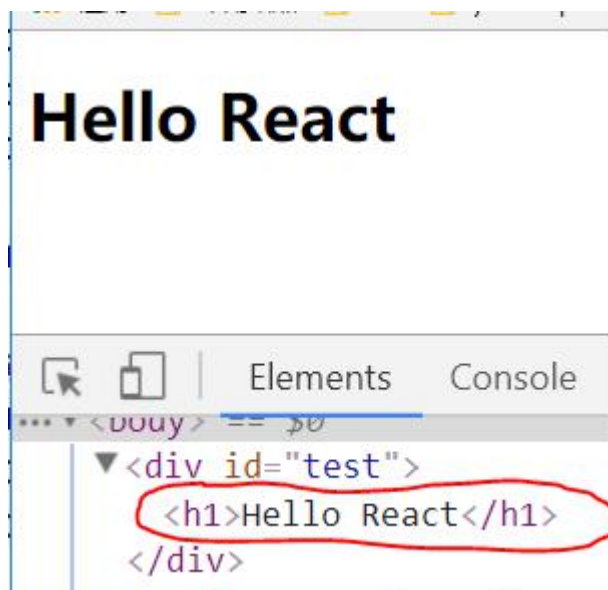
- 1) 虚拟(virtual)DOM, 不总是直接操作 DOM

2) DOM Diff 算法, 最小化页面重绘

1.2. React 的基本使用

注意: 此时只是测试语法使用, 并不是真实项目开发使用

1.2.1. 效果



1.2.2. 相关 js 库

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

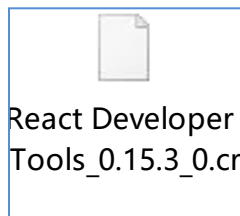
1.2.3. 在页面中导入 js

```
<script type="text/javascript" src="../js/react.development.js"></script>
<script type="text/javascript" src="../js/react-dom.development.js"></script>
<script type="text/javascript" src="../js/babel.min.js"></script>
```

1.2.4. 编码

```
<script type="text/babel"> //必须声明 babel
// 创建虚拟 DOM 元素
const vDom = <h1>Hello React</h1> // 千万不要加引号
// 渲染虚拟 DOM 到页面真实 DOM 容器中
ReactDOM.render(vDom, document.getElementById('test'))
</script>
```

1.2.5. 使用 React 开发者工具调试

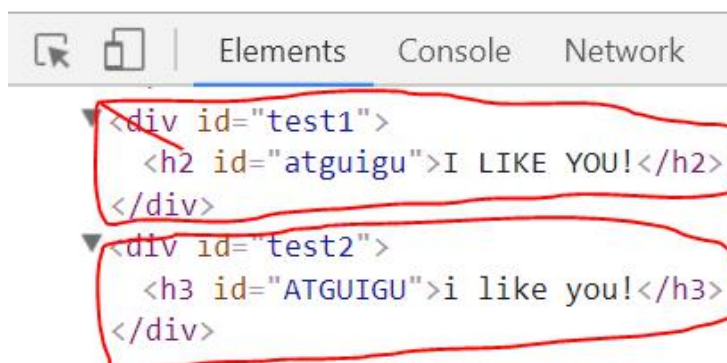


1.3. React JSX

1.3.1. 效果

I LIKE YOU!

i like you!



1.3.2. 虚拟 DOM

- 1) React 提供了一些 API 来创建一种 `特别` 的一般 js 对象
 - a. `var element = React.createElement('h1', {id:'myTitle'}, 'hello')`
 - b. 上面创建的就是一个简单的虚拟 DOM 对象
- 2) 虚拟 DOM 对象最终都会被 React 转换为真实的 DOM
- 3) 我们编码时基本只需要操作 react 的虚拟 DOM 相关数据, react 会转换为真实 DOM 变化而更新界面

1.3.3. JSX

- 1) 全称: JavaScript XML
- 2) react 定义的一种类似于 XML 的 JS 扩展语法: XML+JS

- 3) 作用: 用来创建 react 虚拟 DOM(元素)对象
 - a. `var ele = <h1>Hello JSX!</h1>`
 - b. 注意 1: 它不是字符串, 也不是 HTML/XML 标签
 - c. 注意 2: 它最终产生的就是一个 JS 对象
- 4) 标签名任意: HTML 标签或其它标签
- 5) 标签属性任意: HTML 标签属性或其它
- 6) 基本语法规则
 - a. 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
 - b. 遇到以 `{` 开头的代码, 以 JS 语法解析: 标签中的 js 代码必须用 `{ }` 包含
- 7) babel.js 的作用
 - a. 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
 - b. 只要用了 JSX, 都要加上 `type="text/babel"`, 声明需要 babel 来处理

1.3.4. 渲染虚拟 DOM(元素)

- 1) 语法: `ReactDOM.render(virtualDOM, containerDOM)`
- 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
 - a. 参数一: 纯 js 或 jsx 创建的虚拟 dom 对象
 - b. 参数二: 用来包含虚拟 DOM 元素的真实 dom 元素对象(一般是一个 div)

1.3.5. 建虚拟 DOM 的 2 种方式

- 1) 纯 JS(一般不用)
`React.createElement('h1', {id:'myTitle'}, title)`
- 2) JSX:
`<h1 id='myTitle'>{title}</h1>`

1.3.6. JSX 练习

需求: 动态展示列表数据

前端JS框架列表

- jquery
- zeptoo
- angular
- react全家桶
- vue全家桶

1.4. 模块与组件和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, 一般就是一个 js 文件
- 2) 为什么: js 代码更多更复杂
- 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: 用来实现特定(局部)功能效果的代码集合(html/css/js)
- 2) 为什么: 一个界面的功能更复杂
- 3) 作用: 复用编码, 简化项目编码, 提高运行效率

1.4.3. 模块化

当应用的 js 都以模块来编写的, 这个应用就是一个模块化的应用

1.4.4. 组件化

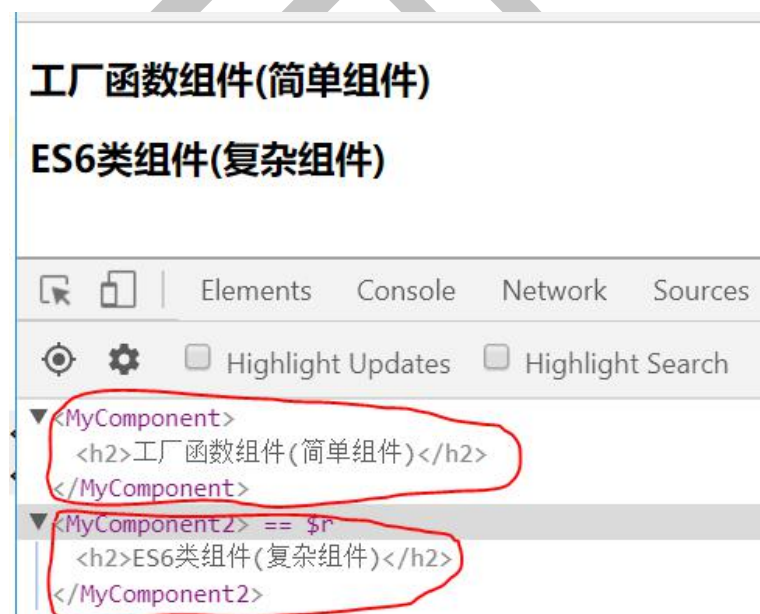
当应用是以多组件的方式实现, 这个应用就是一个组件化的应用



第 2 章: React 面向组件编程

2.1. 基本理解和使用

2.1.1. 效果



2.1.2. 自定义组件(Component):

- 1) 定义组件(2 种方式)

```
/*方式 1: 工厂函数组件(简单组件)*/  
function MyComponent () {  
  return <h2>工厂函数组件(简单组件)</h2>  
}  
  
/*方式 2: ES6 类组件(复杂组件)*/  
class MyComponent2 extends React.Component {  
  render () {  
    console.log(this) // MyComponent2 的实例对象  
    return <h2>ES6 类组件(复杂组件)</h2>  
  }  
}
```

- 2) 渲染组件标签

```
ReactDOM.render(<MyComponent />, document.getElementById('example1'))
```

2.1.3. 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

2.1.4. render()渲染组件标签的基本流程

- 1) React 内部会创建组件实例对象
- 2) 得到包含的虚拟 DOM 并解析为真实 DOM
- 3) 插入到指定的页面元素内部

2.2. 组件三大属性 1: state

2.2.1. 效果



2.2.2. 理解

- 1) state 是组件对象最重要的属性, 值是对象(可以包含多个数据)
- 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(重新渲染组件)

2.2.3. 编码操作

- 1) 初始化状态:

```
constructor (props) {  
  super(props)  
  this.state = {  
    stateProp1 : value1,  
    stateProp2 : value2  
  }  
}
```

- 2) 读取某个状态值

```
this.state.statePropertyName
```

- 3) 更新状态---->组件界面更新

```
this.setState({  
  stateProp1 : value1,  
  stateProp2 : value2  
})
```

2.3. 组件三大属性 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
- 2). 如果性别没有指定，默认为男
- 3). 如果年龄没有指定，默认为18

- 姓名: Tom
- 性别: 女
- 年龄: 18

- 姓名: JACK
- 性别: 男
- 年龄: 17

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

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.propertyName

- 2) 对 props 中的属性值进行类型限制和必要性限制

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

- 3) 扩展属性: 将对象的所有属性通过 props 传递

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

- 4) 默认属性值

```
Person.defaultProps = {  
  name: 'Mary'  
}
```

- 5) 组件类的构造函数

```
constructor (props) {  
  super(props)  
  console.log(props) // 查看所有属性  
}
```

2.3.5. 面试题

问题: 请区别一下组件的 props 和 state 属性

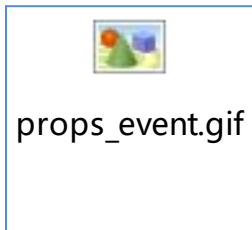
- 1) state: 组件自身内部可变化的数据
- 2) props: 从组件外部向组件内部传递数据, 组件内部只读不修改

2.4. 组件三大属性 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:

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



2.4.2. 组件的 3 大属性之二: refs 属性

- 1) 组件内的标签都可以定义 ref 属性来标识自己
 - a. `<input type="text" ref={input => this.msgInput = input}/>`
 - b. 回调函数在组件初始化渲染完或卸载时自动调用
- 2) 在组件中可以通过 `this.msgInput` 来得到对应的真实 DOM 元素
- 3) 作用: 通过 ref 获取组件内容特定标签对象, 进行读取其相关数据

2.4.3. 事件处理

- 1) 通过 `onXxx` 属性指定组件的事件处理函数(注意大小写)
 - a. React 使用的是自定义(合成)事件, 而不是使用的原生 DOM 事件
 - b. React 中的事件是通过事件委托方式处理的(委托给组件最外层的元素)
- 2) 通过 `event.target` 得到发生事件的 DOM 元素对象

```
<input onFocus={this.handleClick}/>
handleFocus(event) {
  event.target //返回input 对象
}
```

2.4.4. 强烈注意

- 1) 组件内置的方法中的 `this` 为组件对象
- 2) 在组件类中自定义的方法中 `this` 为 null
 - a. 强制绑定 `this`: 通过函数对象的 `bind()`
 - b. 箭头函数(ES6 模块化编码时才能使用)

2.5. 组件的组合

2.5.1. 效果

功能：组件化实现此功能

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



2.5.2. 功能界面的组件化编码流程(无比重要)

- 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
 - a. 动态显示初始化数据
 - b. 交互功能(从绑定事件监听开始)

2.6. 收集表单数据

2.6.1. 效果

需求：自定义包含表单的组件

1. 输入用户名密码后，点击登陆提示输入信息
3. 不提交表单



component
表单.gif

2.6.2. 理解

- 1) 问题: 在 react 应用中, 如何收集表单输入数据
- 2) 包含表单的组件分类
 - a. 受控组件: 表单项输入数据能自动收集成状态
 - b. 非受控组件: 需要时才手动读取表单输入框中的数据

2.7. 组件生命周期

2.7.1. 效果

需求: 自定义组件

1. 让指定的文本做显示/隐藏的渐变动画
2. 切换持续时间为 2S
3. 点击按钮从界面中移除组件界面

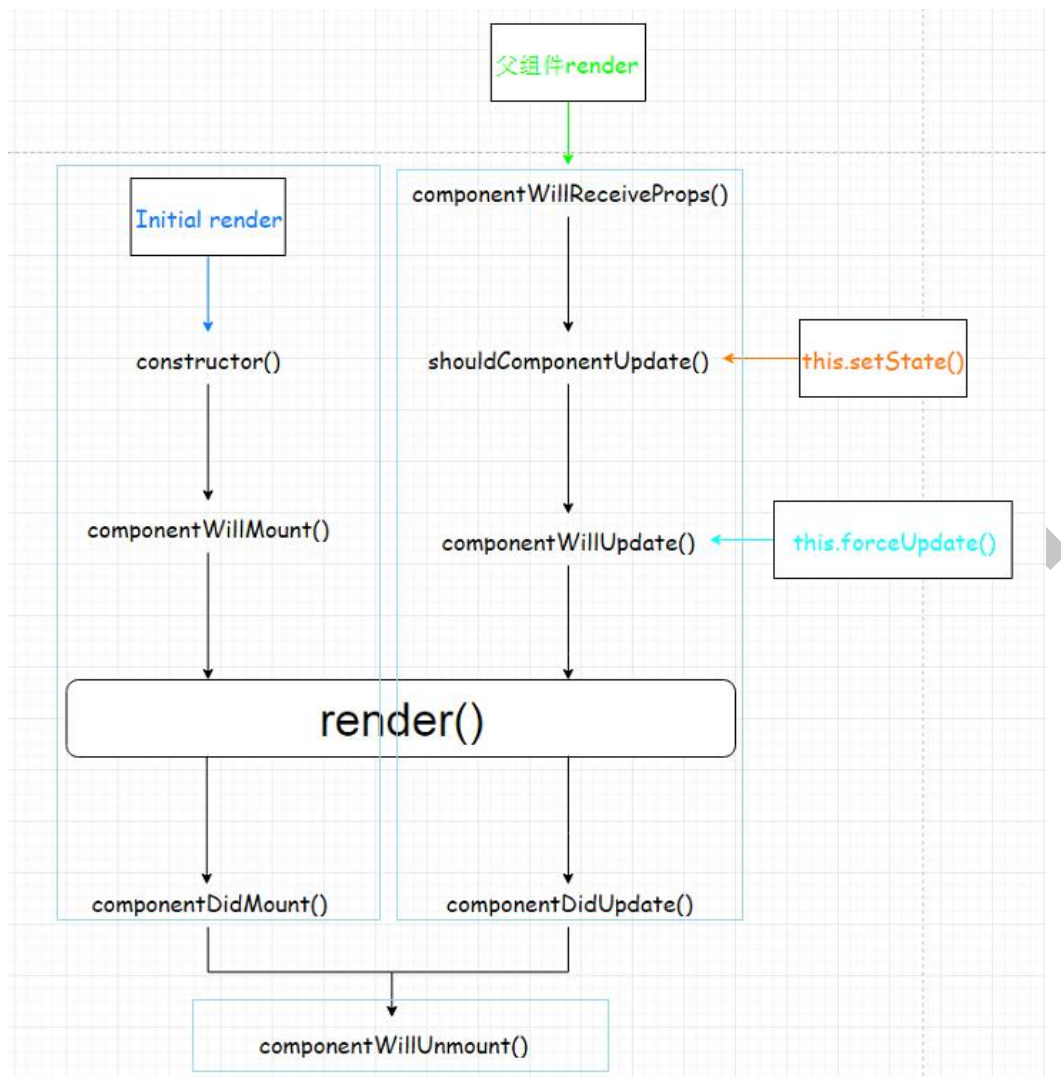


component
生命周期.gif

2.7.2. 理解

- 1) 组件对象从创建到死亡它会经历特定的生命周期阶段
- 2) React 组件对象包含一系列的钩子函数(生命周期回调函数), 在生命周期特定时刻回调
- 3) 我们在定义组件时, 可以重写特定的生命周期回调函数, 做特定的工作

2.7.3. 生命周期流程图



2.7.4. 生命周期详述

- 1) 组件的三个生命周期状态:
 - * Mount: 插入真实 DOM
 - * Update: 被重新渲染
 - * Unmount: 被移出真实 DOM
- 2) React 为每个状态都提供了钩子(hook)函数
 - * componentWillMount()

- * componentDidMount()
- * componentWillUpdate()
- * componentDidUpdate()
- * componentWillUnmount()

3) 生命周期流程:

a. 第一次初始化渲染显示: ReactDOM.render()

- * constructor(): 创建对象初始化 state
- * componentWillMount(): 将要插入回调
- * render(): 用于插入虚拟 DOM 回调
- * componentDidMount(): 已经插入回调

b. 每次更新 state: this.setState()

- * componentWillUpdate(): 将要更新回调
- * render(): 更新(重新渲染)
- * componentDidUpdate(): 已经更新回调

c. 移除组件: ReactDOM.unmountComponentAtNode(containerDom)

- * componentWillUnmount(): 组件将要被移除回调

2.7.5. 重要的钩子

- 1) render(): 初始化渲染或更新渲染调用
- 2) componentDidMount(): 开启监听, 发送 ajax 请求
- 3) componentWillUnmount(): 做一些收尾工作, 如: 清理定时器
- 4) componentWillReceiveProps(): 后面需要时讲

2.8. 虚拟 DOM 与 DOM Diff 算法

2.8.1. 效果



component
虚拟DOM.gif

```
class HelloWorld extends React.Component {
  constructor(props) {
    super(props)
    this.state = {
      date: new Date()
    }
  }

  componentDidMount () {
    setInterval(() => {
      this.setState({
        date: new Date()
      })
    }, 1000)
  }

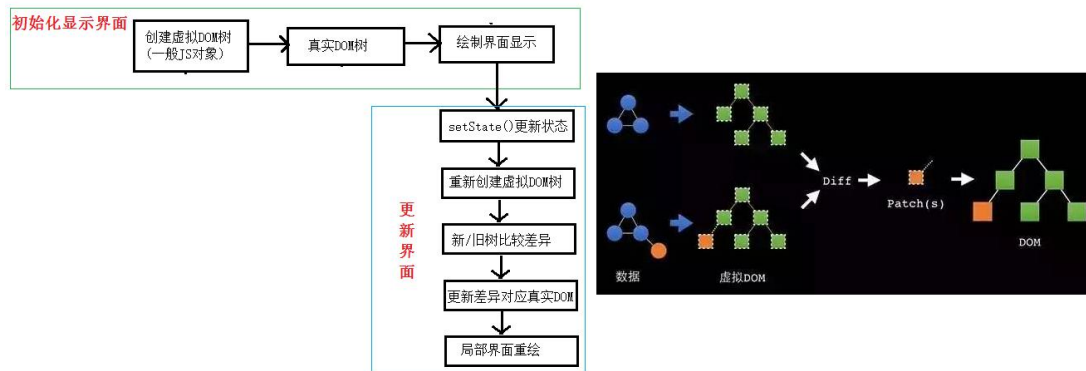
  render () {
    console.log('render()')
    return (
      <p>
        Hello, <input type="text" placeholder="Your name here"/>!&nbsp;
        It is {this.state.date.toTimeString()}
      </p>
    )
  }
}

ReactDOM.render(
  <HelloWorld/>,

```

```
document.getElementById('example')  
)
```

2.8.2. 基本原理图



第 3 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
 - a. 包含了所有需要的配置
 - b. 指定好了所有的依赖
 - c. 可以直接安装/编译/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) 项目的整体技术架构为: 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

3.1.2. 创建项目并启动

```
npm install -g create-react-app
create-react-app hello-react
cd hello-react
npm start
```

3.1.3. react 脚手架项目结构

```
ReactNews
|--node_modules---第三方依赖模块文件夹
|--public
|   |--index.html-----主页面
|--scripts
|   |--build.js-----build 打包引用配置
|   |--start.js-----start 运行引用配置
|--src-----源码文件夹
|   |--components-----react 组件
|   |--index.js-----应用入口 js
|--.gitignore-----git 版本管制忽略的配置
|--package.json----应用包配置文件
|--README.md-----应用描述说明的 readme 文件
```

3.2. demo: 评论管理

3.2.1. 效果



demo_comment.
gif

3.2.2. 拆分组件

应用组件: App

- * state: comments/array

添加评论组件: CommentAdd

- * state: username/string, content/string

- * props: add/func

评论列表组件: CommentList

- * props: comment/object, delete/func, index/number

评论项组件: CommentItem

- * props: comments/array, delete/func

3.2.3. 实现静态组件

3.2.4. 实现动态组件

动态展示初始化数据

- * 初始化状态数据
- * 传递属性数据

响应用户操作, 更新组件界面

- * 绑定事件监听, 并处理
- * 更新 state

第 4 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 请求与后台进行交互(json 数据)
- 3) react 应用中需要集成第三方 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
 - a. 封装 XMLHttpRequest 对象的 ajax
 - b. promise 风格
 - c. 可以用在浏览器端和 node 服务器端
- 3) fetch: 原生函数, 但老版本浏览器不支持
 - a. 不再使用 XMLHttpRequest 对象提交 ajax 请求
 - b. 为了兼容低版本的浏览器, 可以引入兼容库 fetch.js

4.1.3. 效果

需求:

1. 界面效果如下
2. 根据指定的关键字在 github 上搜索匹配的最受关注的库
3. 显示库名, 点击链接查看库
4. 测试接口: <https://api.github.com/search/repositories?q=r&sort=stars>



ajax.gif

4.2. 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=12345')
  .then(function (response) {
    console.log(response);
  })
  .catch(function (error) {
    console.log(error);
  });
```

```
axios.get('/user', {
  params: {
    ID: 12345
  }
})
  .then(function (response) {
    console.log(response);
  })
  .catch(function (error) {
    console.log(error);
  });
```

2) POST 请求

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

4.3. Fetch

4.3.1. 文档

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

4.3.2. 相关 API

- 1) GET 请求

```
fetch(url).then(function(response) {  
  return response.json()  
}).then(function(data) {  
  console.log(data)  
}).catch(function(e) {  
  console.log(e)  
});
```

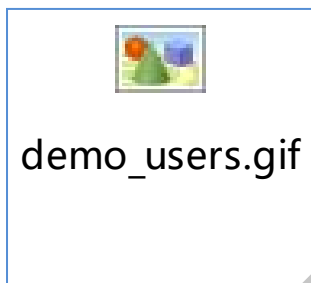
- 2) POST 请求

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

```
}).catch(function(e) {  
  console.log(e)  
})
```

4.4. demo: github users

4.4.1. 效果



4.4.2. 拆分组件

App
* state: searchName/string
Search
* props: setSearchName/func
List
* props: searchName/string
* state: firstView/bool, loading/bool, users/array, errMsg/string

4.4.3. 编写静态组件

4.4.4. 编写动态组件

componentWillReceiveProps(nextProps): 监视接收到新的 props, 发送 ajax

使用 axios 库发送 ajax 请求

第 5 章：几个重要技术总结

5.1. 组件间通信

5.1.1. 方式一：通过 props 传递

- 1) 共同的数据放在父组件上, 特有的数据放在自己组件内部(state)
- 2) 通过 props 可以传递一般数据和函数数据, 只能一层一层传递
- 3) 一般数据-->父组件传递数据给子组件-->子组件读取数据
- 4) 函数数据-->子组件传递数据给父组件-->子组件调用函数

5.1.2. 方式二：使用消息订阅(subscribe)-发布(publish)机制

- 1) 工具库: PubSubJS
- 2) 下载: `npm install pubsub-js --save`
- 3) 使用:

```
import PubSub from 'pubsub-js' //引入
PubSub.subscribe('delete', function(data){ }); //订阅
PubSub.publish('delete', data) //发布消息
```

5.1.3. 方式三: redux

后面专门讲解

5.2. 事件监听理解

5.2.1. 原生 DOM 事件

- 1) 绑定事件监听
 - a. 事件名(类型): 只有有限的几个, 不能随便写

- b. 回调函数
- 2) 触发事件
 - a. 用户操作界面
 - b. 事件名(类型)
 - c. 数据()

5.2.2. 自定义事件(消息机制)

- 1) 绑定事件监听
 - a. 事件名(类型): 任意
 - b. 回调函数: 通过形参接收数据, 在函数体处理事件
- 2) 触发事件(编码)
 - a. 事件名(类型): 与绑定的事件监听的事件名一致
 - b. 数据: 会自动传递给回调函数

5.3. ES6 常用新语法

- 1) 定义常量/变量: `const/let`
- 2) 解构赋值: `let {a, b} = this.props` `import {aa} from 'xxx'`
- 3) 对象的简洁表达: `{a, b}`
- 4) 箭头函数:
 - a. 常用场景
 - * 组件的自定义方法: `xxx = () => {}`
 - * 参数匿名函数
 - b. 优点:
 - * 简洁
 - * 没有自己的 `this`, 使用引用 `this` 查找的是外部 `this`
- 5) 扩展(三点)运算符: 拆解对象(`const MyProps = {}, <Xxx {...MyProps}>`)
- 6) 类: `class/extends/constructor/super`
- 7) ES6 模块化: `export default | import`

第 6 章：react-router4

6.1. 相关理解

6.1.1. react-router 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

6.1.2. SPA 的理解

- 1) 单页 Web 应用 (single page web 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, 本身也不会向服务器发请求
- 4) 当点击路由链接时, 只会做页面的局部更新
- 5) 数据都需要通过 ajax 请求获取, 并在前端异步展现

6.1.3. 路由的理解

- 1) 什么是路由?
 - a. 一个路由就是一个映射关系(key:value)
 - b. key 为路由路径, value 可能是 function/component
- 2) 路由分类
 - a. 后台路由: node 服务器端路由, value 是 function, 用来处理客户端提交的请求并返回一个响应数据

- b. 前台路由: 浏览器端路由, value 是 component, 当请求的是路由 path 时, 浏览器端前没有发送 http 请求, 但界面会更新显示对应的组件
- 3) 后台路由
 - a. 注册路由: `router.get(path, function(req, res))`
 - b. 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
- 4) 前端路由
 - a. 注册路由: `<Route path="/about" component={About}>`
 - b. 当浏览器的 hash 变为 #about 时, 当前路由组件就会变为 About 组件

6.1.4. 前端路由的实现

- 1) history 库
 - a. 网址: <https://github.com/ReactTraining/history>
 - b. 管理浏览器会话历史(history)的工具库
 - c. 包装的是原生 BOM 中 window.history 和 window.location.hash
- 2) history API
 - a. `History.createBrowserHistory()`: 得到封装 window.history 的管理对象
 - b. `History.createHashHistory()`: 得到封装 window.location.hash 的管理对象
 - c. `history.push()`: 添加一个新的历史记录
 - d. `history.replace()`: 用一个新的历史记录替换当前的记录
 - e. `history.goBack()`: 回退到上一个历史记录
 - f. `history.goForward()`: 前进到下一个历史记录
 - g. `history.listen(function(location){})`: 监视历史记录的变化
- 3) 测试



history-方式1.gif



history-方式2.gif

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>history test</title>
</head>
<body>
  <p><input type="text"></p>
  <a href="/test1" onclick="return push('/test1')">test1</a><br><br>
  <button onClick="push('/test2')">push test2</button><br><br>
  <button onClick="back()">回退</button><br><br>
  <button onClick="forward()">前进</button><br><br>
  <button onClick="replace('/test3')">replace test3</button><br><br>

  <script type="text/javascript"
src="https://cdn.bootcss.com/history/4.7.2/history.js"></script>
  <script type="text/javascript">
    let history = History.createBrowserHistory() // 方式一
    // history = History.createHashHistory() // 方式二
    // console.log(history)

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

    function back() {
      history.goBack()
    }

    function forward() {
      history.goForward()
    }

    function replace (to) {
      history.replace(to)
    }

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

```
</script>  
</body>  
</html>
```

6.2. react-router 相关 API

6.2.1. 组件

- 1) <BrowserRouter>
- 2) <HashRouter>
- 3) <Route>
- 4) <Redirect>
- 5) <Link>
- 6) <NavLink>
- 7) <Switch>

6.2.2. 其它

- 1) history 对象
- 2) match 对象
- 3) withRouter 函数

6.3. 基本路由使用

6.3.1. 效果



react-router
demo1.gif

6.3.2. 准备

- 1) 下载 react-router: `npm install --save react-router@4`
- 2) 引入 bootstrap.css: `<link rel="stylesheet" href="/css/bootstrap.css">`

6.3.3. 路由组件: views/about.jsx

```
import React from 'react'
export default function About() {
  return <div>About 组件内容</div>
}
```

6.3.4. 路由组件: views/home.jsx

```
import React from 'react'
export default function About() {
  return <div>Home 组件内容</div>
}
```

6.3.5. 包装 NavLink 组件: components/my-nav-link.jsx

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

export default function MyNavLink(props) {
  return <NavLink {...props} activeClassName='activeClass' />
}
```

6.3.6. 应用组件: components/app.jsx

```
import React from 'react'
import {Route, Switch, Redirect} from 'react-router-dom'
import MyNavLink from './components/my-nav-link'
import About from './views/about'
import Home from './views/home'

export default class App extends React.Component {

  render () {
    return (
      <div>

        <div className="row">
          <div className="col-xs-offset-2 col-xs-8">
            <div className="page-header">
              <h2>React Router Demo</h2>
            </div>
          </div>
        </div>

        <div className="row">
          <div className="col-xs-2 col-xs-offset-2">
            <div className="list-group">
              /* 导航路由链接*/
              <MyNavLink className="list-group-item" to="/about" >About</MyNavLink>
              <MyNavLink className="list-group-item" to="/home" >Home</MyNavLink>
            </div>
          </div>
          <div className="col-xs-6">
            <div className="panel">
              <div className="panel-body">
                /* 可切换的路由组件*/
                <Switch>
                  <Route path="/about" component={About} />
                  <Route path="/home" component={Home} />
                  <Redirect to="/about" />
                </Switch>
              </div>
            </div>
          </div>
        </div>
      </div>
    )
  }
}
```



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

6.3.7. 自定义样式: index.css

```
.activeClass {
  color: red !important;
}
```

6.3.8. 入口 JS: index.js

```
import React from 'react'
import ReactDOM from 'react-dom'
import {BrowserRouter, HashRouter} from 'react-router-dom'
import App from './components/app'

import './index.css'

ReactDOM.render(
  (
    <BrowserRouter>
      <App />
    </BrowserRouter>
    /*<HashRouter>
      <App />
    </HashRouter>*/
  ),
  document.getElementById('root')
)
```

6.4. 嵌套路由使用

6.4.1. 效果



react-router
demo2.gif

6.4.2. 二级路由组件: views/news.jsx

```
import React from 'react'
export default class News extends React.Component {
  state = {
    newsArr: ['news001', 'news002', 'news003']
  }

  render () {
    return (
      <div>
        <ul>
          {
            this.state.newsArr.map((news, index) => <li key={index}>{news}</li>)
          }
        </ul>
      </div>
    )
  }
}
```

6.4.3. 二级路由组件: views/message.jsx

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

```
export default class Message extends React.Component {  
  state = {  
    messages: []  
  }  
  
  componentDidMount () {  
    // 模拟发送 ajax 请求  
    setTimeout(() => {  
      const data = [  
        {id: 1, title: 'Message001'},  
        {id: 3, title: 'Message003'},  
        {id: 6, title: 'Message006'},  
      ]  
      this.setState({  
        messages: data  
      })  
    }, 1000)  
  }  
  
  render () {  
    const path = this.props.match.path  
  
    return (  
      <div>  
        <ul>  
          {  
            this.state.messages.map((m, index) => {  
              return (  
                <li key={index}>  
                  <Link to='???'>{m.title}</Link>  
                </li>  
              )  
            })  
          }  
        </ul>  
      </div>  
    )  
  }  
}
```

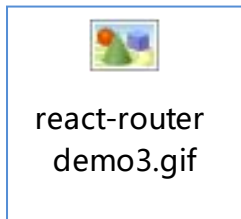
6.4.4. 一级路由组件: views/home.jsx

```
import React from 'react'
import {Switch, Route, Redirect} from 'react-router-dom'
import MyNavLink from './components/my-nav-link'
import News from './views/news'
import Message from './views/message'

export default function Home() {
  return (
    <div>
      <h2>Home 组件内容</h2>
      <div>
        <ul className="nav nav-tabs">
          <li>
            <MyNavLink to="/home/news">News</MyNavLink>
          </li>
          <li>
            <MyNavLink to="/home/message">Message</MyNavLink>
          </li>
        </ul>
        <Switch>
          <Route path="/home/news" component={News} />
          <Route path="/home/message" component={Message} />
          <Redirect to="/home/news"/>
        </Switch>
      </div>
    </div>
  )
}
```

6.5. 向路由组件传递参数数据

6.5.1. 效果



6.5.2. 三级路由组件: views/message-detail.jsx

```
import React from 'react'
const messageDetails = [
  {id: 1, title: 'Message001', content: '我爱你, 中国'},
  {id: 3, title: 'Message003', content: '我爱你, 老婆'},
  {id: 6, title: 'Message006', content: '我爱你, 孩子'},
]
export default function MessageDetail(props) {

  const id = props.match.params.id
  const md = messageDetails.find(md => md.id===id*1)

  return (
    <ul>
      <li>ID: {md.id}</li>
      <li>TITLE: {md.title}</li>
      <li>CONTENT: {md.content}</li>
    </ul>
  )
}
```

6.5.3. 二级路由组件: views/message.jsx

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

```
import MessageDetail from "../views/message-detail"

export default class Message extends React.Component {
  state = {
    messages: []
  }

  componentDidMount () {
    // 模拟发送ajax 请求
    setTimeout(() => {
      const data = [
        {id: 1, title: 'Message001'},
        {id: 3, title: 'Message003'},
        {id: 6, title: 'Message006'},
      ]
      this.setState({
        messages: data
      })
    }, 1000)
  }

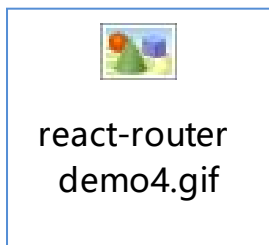
  render () {
    const path = this.props.match.path

    return (
      <div>
        <ul>
          {
            this.state.messages.map((m, index) => {
              return (
                <li key={index}>
                  <Link to={` ${path}/${m.id}`}>{m.title}</Link>
                </li>
              )
            })
          }
        </ul>
        <hr/>
        <Route path={` ${path}/:id`} component={MessageDetail}></Route>
      </div>
    )
  }
}
```

```
}  
}
```

6.6. 多种路由跳转方式

6.6.1. 效果



6.6.2. 二级路由: views/message.jsx

```
import React from 'react'  
import {Link, Route} from 'react-router-dom'  
import MessageDetail from "../views/message-detail"  
  
export default class Message extends React.Component {  
  state = {  
    messages: []  
  }  
  
  componentDidMount () {  
    // 模拟发送 ajax 请求  
    setTimeout(() => {  
      const data = [  
        {id: 1, title: 'Message001'},  
        {id: 3, title: 'Message003'},  
        {id: 6, title: 'Message006'},  
      ]  
      this.setState({  
        messages: data  
      })  
    }, 1000)
```

```
}

ShowDetail = (id) => {
  this.props.history.push(`/home/message/${id}`)
}

ShowDetail2 = (id) => {
  this.props.history.replace(`/home/message/${id}`)
}

back = () => {
  this.props.history.goBack()
}

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

render () {
  const path = this.props.match.path

  return (
    <div>
      <ul>
        {
          this.state.messages.map((m, index) => {
            return (
              <li key={index}>
                <Link to={`/${path}/${m.id}`}>{m.title}</Link>
                &nbsp;
                <button onClick={() => this.ShowDetail(m.id)}>查看详情
(push)</button>&nbsp;
                <button onClick={() => this.ShowDetail2(m.id)}>查看详情
(replace)</button>
              </li>
            )
          })
        }
      </ul>
      <p>
        <button onClick={this.back}>返回</button>&nbsp;

```



```
        <button onClick={this.forward}>前进</button>&nbsp;  </p>
    </hr/>
    <Route path={` ${path}/${id}`} component={MessageDetail}></Route>
  </div>
)
}
}
```

第 7 章：react-ui

7.1. 最流行的开源 React UI 组件库

7.1.1. material-ui(国外)

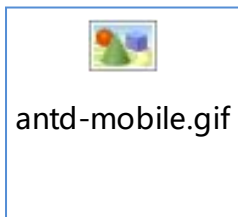
- 1) 官网: <http://www.material-ui.com/#/>
- 2) github: <https://github.com/callemall/material-ui>

7.1.2. ant-design(国内蚂蚁金服)

- 1) PC 官网: <https://ant.design/index-cn>
- 2) 移动官网: <https://mobile.ant.design/index-cn>
- 3) Github: <https://github.com/ant-design/ant-design/>
- 4) Github: <https://github.com/ant-design/ant-design-mobile/>

7.2. ant-design-mobile 使用入门

7.2.1. 效果



7.2.2. 使用 create-react-app 创建 react 应用

```
npm install create-react-app -g
create-react-app antm-demo
cd antm-demo
npm start
```

7.2.3. 搭建 antd-mobile 的基本开发环境

1) 下载

```
npm install antd-mobile --save
```

2) src/App.jsx

```
import React, {Component} from 'react'
// 分别引入需要使用的组件
import Button from 'antd-mobile/lib/button'
import Toast from 'antd-mobile/lib/toast'

export default class App extends Component {
  handleClick = () => {
    Toast.info('提交成功', 2)
  }

  render() {
    return (
      <div>
        <Button type="primary" onClick={this.handleClick}>提交</Button>
      </div>
    )
  }
}
```

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

3) src/index.js

```
import React from 'react';
import ReactDOM from 'react-dom'
import App from './App'
// 引入整体css
import 'antd-mobile/dist/antd-mobile.css'

ReactDOM.render(<App />, document.getElementById('root'))
```

4) index.html

```
<meta name="viewport" content="width=device-width, initial-scale=1, maximum-scale=1,
minimum-scale=1, user-scalable=no" />

<script
src="https://as.alipayobjects.com/g/component/fastclick/1.0.6/fastclick.js"></scrip
t>
<script>
  if ('addEventListener' in document) {
    document.addEventListener('DOMContentLoaded', function() {
      FastClick.attach(document.body);
    }, false);
  }
  if(!window.Promise) {
    document.writeln('<script
src="https://as.alipayobjects.com/g/component/es6-promise/3.2.2/es6-promise.min.js"
'+>'+<'+/'+'script>');
  }
</script>
```

7.2.4. 实现按需打包(组件 js/css)

1) 下载依赖包

```
yarn add react-app-rewired --dev  
yarn add babel-plugin-import --dev
```

2) 修改默认配置:

● package.json

```
"scripts": {  
  "start": "react-app-rewired start",  
  "build": "react-app-rewired build",  
  "test": "react-app-rewired test --env=jsdom"  
}
```

● config-overrides.js

```
const {injectBabelPlugin} = require('react-app-rewired');  
module.exports = function override(config, env) {  
  config = injectBabelPlugin(['import', {libraryName: 'antd-mobile', style: 'css'}],  
    config);  
  return config;  
};
```

3) 编码

```
// import 'antd-mobile/dist/antd-mobile.css'  
  
// import Button from 'antd-mobile/lib/button'  
// import Toast from 'antd-mobile/lib/toast'  
import {Button, Toast} from 'antd-mobile'
```

第 8 章：redux

8.1. redux 理解

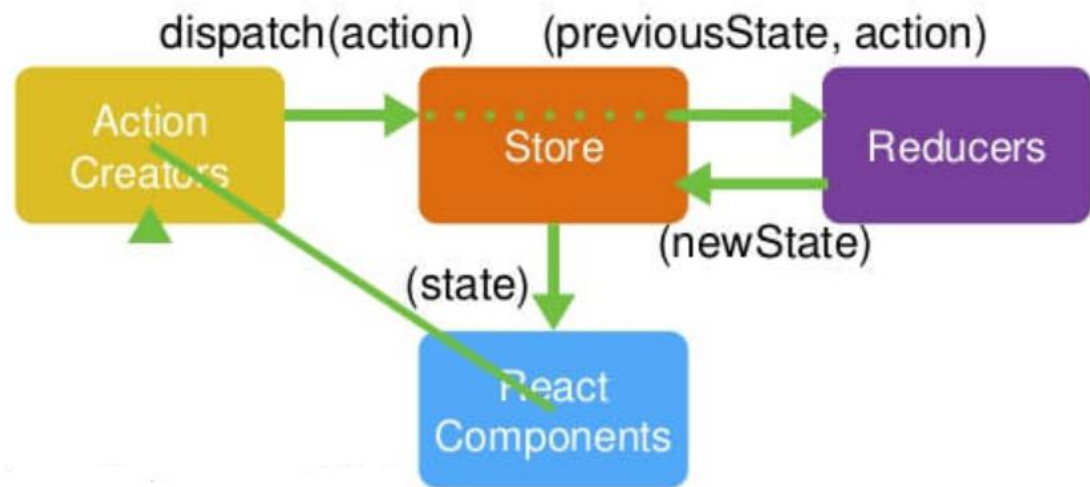
8.1.1. 学习文档

- 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>

8.1.2. redux 是什么?

- 1) redux 是一个独立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

8.1.3. redux 工作流程



8.1.4. 什么情况下需要使用 redux

- 1) 总体原则: 能不用就不用, 如果不用比较吃力才考虑使用
- 2) 某个组件的状态, 需要共享
- 3) 某个状态需要在任何地方都可以拿到
- 4) 一个组件需要改变全局状态
- 5) 一个组件需要改变另一个组件的状态

8.2. redux 的核心 API

8.2.1. createStore()

- 1) 作用:
创建包含指定 reducer 的 store 对象
- 2) 编码:

```
import {createStore} from 'redux'  
import counter from './reducers/counter'  
const store = createStore(counter)
```

8.2.2. 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:
getState()
dispatch(action)
subscribe(listener)
- 4) 编码:

```
store.getState()  
store.dispatch({type:'INCREMENT', number})
```

```
store.subscribe(render)
```

8.2.3. applyMiddleware()

1) 作用:

应用上基于 `redux` 的中间件(插件库)

2) 编码:

```
import {createStore, applyMiddleware} from 'redux'
import thunk from 'redux-thunk' // redux 异步中间件
const store = createStore(
  counter,
  applyMiddleware(thunk) // 应用上异步中间件
)
```

8.2.4. combineReducers()

1) 作用:

合并多个 `reducer` 函数

2) 编码:

```
export default combineReducers({
  user,
  chatUser,
  chat
})
```

8.3. redux 的三个核心概念

8.3.1. action

1) 标识要执行行为的对象

2) 包含 2 个方面的属性

a. `type`: 标识属性, 值为字符串, 唯一, 必要属性

b. `xxx`: 数据属性, 值类型任意, 可选属性

3) 例子:

```
const action = {  
  type: 'INCREMENT',  
  data: 2  
}
```

- 4) Action Creator(创建 Action 的工厂函数)

```
const increment = (number) => ({type: 'INCREMENT', data: number})
```

8.3.2. reducer

- 1) 根据老的 state 和 action, 产生新的 state 的纯函数
- 2) 样例

```
export default function counter(state = 0, action) {  
  switch (action.type) {  
    case 'INCREMENT':  
      return state + action.data  
    case 'DECREMENT':  
      return state - action.data  
    default:  
      return state  
  }  
}
```

- 3) 注意

- a. 返回一个新的状态
- b. 不要修改原来的状态

8.3.3. store

- 1) 将 state, action 与 reducer 联系在一起的对象
- 2) 如何得到此对象?

```
import {createStore} from 'redux'  
import reducer from './reducers'  
const store = createStore(reducer)
```

- 3) 此对象的功能?

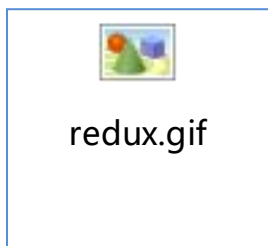
getState(): 得到 state

dispatch(action): 分发 action, 触发 reducer 调用, 产生新的 state

subscribe(listener): 注册监听, 当产生了新的 state 时, 自动调用

8.4. 使用 redux 编写应用

8.4.1. 效果



8.4.2. 下载依赖包

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

8.4.3. redux/action-types.js

```
/*  
action 对象的 type 常量名称模块  
*/  
export const INCREMENT = 'increment'  
export const DECREMENT = 'decrement'
```

8.4.4. redux/actions.js

```
/*  
action creator 模块  
*/  
import {INCREMENT, DECREMENT} from './action-types'  
  
export const increment = number => ({type: INCREMENT, number})  
export const decrement = number => ({type: DECREMENT, number})
```

8.4.5. redux/reducers.js

```
/*
根据老的 state 和指定 action, 处理返回一个新的 state
*/
import {INCREMENT, DECREMENT} from '../constants/ActionTypes'

import {INCREMENT, DECREMENT} from './action-types'

export function counter(state = 0, action) {
  console.log('counter', state, action)
  switch (action.type) {
    case INCREMENT:
      return state + action.number
    case DECREMENT:
      return state - action.number
    default:
      return state
  }
}
```

8.4.6. components/app.jsx

```
/*
应用组件
*/
import React, {Component} from 'react'
import PropTypes from 'prop-types'
import * as actions from '../redux/actions'

export default class App extends Component {

  static propTypes = {
    store: PropTypes.object.isRequired,
  }

  increment = () => {
    const number = this.refs.numSelect.value * 1
    this.props.store.dispatch(actions.increment(number))
  }
}
```

```
}

decrement = () => {
  const number = this.refs.numSelect.value * 1
  this.props.store.dispatch(actions.decrement(number))
}

incrementIfOdd = () => {
  const number = this.refs.numSelect.value * 1

  let count = this.props.store.getState()
  if (count % 2 === 1) {
    this.props.store.dispatch(actions.increment(number))
  }
}

incrementAsync = () => {
  const number = this.refs.numSelect.value * 1
  setTimeout(() => {
    this.props.store.dispatch(actions.increment(number))
  }, 1000)
}

render() {
  return (
    <div>
      <p>
        click {this.props.store.getState()} times {' '}
      </p>
      <select ref="numSelect">
        <option value="1">1</option>
        <option value="2">2</option>
        <option value="3">3</option>
      </select>{' '}
      <button onClick={this.increment}>+</button>
      {' '}
      <button onClick={this.decrement}>-</button>
      {' '}
      <button onClick={this.incrementIfOdd}>increment if odd</button>
      {' '}
      <button onClick={this.incrementAsync}>increment async</button>
    </div>
  )
}
```

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

8.4.7. index.js

```
import React from 'react'
import ReactDOM from 'react-dom'
import { createStore } from 'redux'

import App from './components/app'
import { counter } from './redux/reducers'

// 根据 counter 函数创建 store 对象
const store = createStore(counter)

// 定义渲染根组件标签的函数
const render = () => {
  ReactDOM.render(
    <App store={store}/>,
    document.getElementById('root')
  )
}

// 初始化渲染
render()

// 注册(订阅)监听, 一旦状态发生改变, 自动重新渲染
store.subscribe(render)
```

8.4.8. 问题

- 1) redux 与 react 组件的代码耦合度太高
- 2) 编码不够简洁

8.5. react-redux

8.5.1. 理解

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

8.5.2. React-Redux 将所有组件分成两大类

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

8.5.3. 相关 API

- 1) Provider

让所有组件都可以得到 state 数据

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

- 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

```
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

简洁语法可以直接指定为 actions 对象或包含多个 action 方法的对象

8.5.4. 使用 react-redux

1) 下载依赖包

```
npm install --save react-redux
```

2) redux/action-types.js

不变

3) redux/actions.js

不变

4) redux/reducers.js

不变

5) components/counter.jsx

```
/*  
UI 组件: 不包含任何 redux API  
*/  
import React from 'react'  
import PropTypes from 'prop-types'  
  
export default class Counter extends React.Component {  
  
  static propTypes = {  
    count: PropTypes.number.isRequired,  
    increment: PropTypes.func.isRequired,  
    decrement: PropTypes.func.isRequired  
  }  
}
```

```
increment = () => {
  const number = this.refs.numSelect.value * 1
  this.props.increment(number)
}

decrement = () => {
  const number = this.refs.numSelect.value * 1
  this.props.decrement(number)
}

incrementIfOdd = () => {
  const number = this.refs.numSelect.value * 1
  let count = this.props.count
  if (count % 2 === 1) {
    this.props.increment(number)
  }
}

incrementAsync = () => {
  const number = this.refs.numSelect.value * 1
  setTimeout(() => {
    this.props.increment(number)
  }, 1000)
}

render() {
  return (
    <div>
      <p>
        click {this.props.count} times {' '}
      </p>
      <select ref="numSelect">
        <option value="1">1</option>
        <option value="2">2</option>
        <option value="3">3</option>
      </select>{' '}
      <button onClick={this.increment}></button>
      {' '}
      <button onClick={this.decrement}></button>
      {' '}
    </div>
  )
}
```

```
    <button onClick={this.incrementIfOdd}>increment if odd</button>
    { ' ' }
    <button onClick={this.incrementAsync}>increment async</button>
  </div>
)
}
}
```

6) containers/app.jsx

```
/*
包含 Counter 组件的容器组件
*/
import React from 'react'
// 引入连接函数
import {connect} from 'react-redux'
// 引入 action 函数
import {increment, decrement} from '../redux/actions'

import Counter from '../components/counter'

// 向外暴露连接App 组件的包装组件
export default connect(
  state => ({count: state}),
  {increment, decrement}
)(Counter)
```

7) index.js

```
import React from 'react'
import ReactDOM from 'react-dom'
import {createStore} from 'redux'
import {Provider} from 'react-redux'

import App from './containers/app'
import {counter} from '../redux/reducers'

// 根据 counter 函数创建 store 对象
const store = createStore(counter)

// 定义渲染根组件标签的函数
ReactDOM.render(
  (
```



```
<Provider store={store}>
  <App />
</Provider>
),
document.getElementById('root')
```

8.5.5. 问题

- 1) redux 默认是不能进行异步处理的,
- 2) 应用中又需要在 redux 中执行异步任务(ajax, 定时器)

8.6. redux 异步编程

8.6.1. 下载 redux 插件(异步中间件)

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

8.6.2. index.js

```
import {createStore, applyMiddleware} from 'redux'
import thunk from 'redux-thunk'
// 根据 counter 函数创建 store 对象
const store = createStore(
  counter,
  applyMiddleware(thunk) // 应用上异步中间件
)
```

8.6.3. redux/actions.js

```
// 异步 action creator( 返回一个函数)
```

```
export const incrementAsync = number => {  
  return dispatch => {  
    setTimeout(() => {  
      dispatch(increment(number))  
    }, 1000)  
  }  
}
```

8.6.4. components/counter.jsx

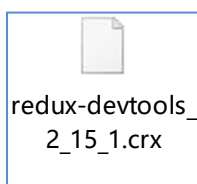
```
incrementAsync = () => {  
  const number = this.refs.numSelect.value*1  
  this.props.incrementAsync(number)  
}
```

8.6.5. containers/app.jsx

```
import {increment, decrement, incrementAsync} from '../redux/actions'  
// 向外暴露连接App 组件的包装组件  
export default connect(  
  state => ({count: state}),  
  {increment, decrement, incrementAsync}  
) (Counter)
```

8.7. 使用上 redux 调试工具

8.7.1. 安装 chrome 浏览器插件



8.7.2. 下载工具依赖包

```
npm install --save-dev redux-devtools-extension
```

8.7.3. 编码

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

const store = createStore(
  counter,
  composeWithDevTools(applyMiddleware(thunk))
)
```

8.8. 相关重要知识：纯函数和高阶函数

8.8.1. 纯函数

- 1) 一类特别的函数：只要是同样的输入，必定得到同样的输出
- 2) 必须遵守以下一些约束
 - a. 不得改写参数
 - b. 不能调用系统 I/O 的 API
 - c. 能调用 `Date.now()` 或者 `Math.random()` 等不纯的方法
- 3) `reducer` 函数必须是一个纯函数

8.8.2. 高阶函数

- 4) 理解：一类特别的函数
 - a. 情况 1: 参数是函数
 - b. 情况 2: 返回是函数

- 5) 常见的高阶函数:
 - a. 定时器设置函数
 - b. 数组的 `map()/filter()/reduce()/find()/bind()`
 - c. react-redux 中的 `connect` 函数
- 6) 作用:
 - a. 能实现更加动态, 更加可扩展的功能