**React**

**初识React**

**JavaScript新特性**

**JavaScript函数式编程**

**React基础**

**脚手架安装**

第一种方式：

1. npm install -g create-react-app;

2. create-react-app my-app;

3. cd my-app;

4. npm start;

第二种方式：

1. npx create-react-app my-app;

2. cd my-app;

3. npm start;

**JSX**

**JSX简介定义**

作为UI描述和UI数据之间的桥梁,让UI结构直观清晰，同时因为JSX本质上仍是JavaScript，所以可以使用Js语法

**JSX语法**

**1.** 基本语法：使用成对的标签构成一个树状结构的数据;

**2.** 标签类型：DOM类型的标签(div,span等)、React组件类型的标签;DOM类型的标签首字母必须是小写，React组件类型的标签首字母必须是大写，这也是区分两种类型标签的依据;二者可以互相嵌套使用;

**3. JavaScrip**t表达式：在JSX中使用JavaScript表达式需要将表达式用大括号“{}”包起来，主要使用场景有：通过表达式给标签属性赋值、通过表达式定义**子组**件;注意：JSX中只能使用JavaScript表达式，不能使用多行JavaScript语句，可以使用三目运算符和逻辑与运算符(可以代替if语句);

**4.** 标签属性：当JSX标签是DOM类型的标签时，对应的DOM标签支持的属性JSX也支持，部分会改变成驼峰式命名;当JSX标签是React组件类型时，可以任意自定义标签的属性名;

**5**. 注释：JSX中的注释需要用大括号“{}”将“/\*\*/”包裹起来({/\*这是注释\*/});

**JSX不是必需的**

JSX语法对使用React不是必需的，实际上，JSX语法只是React.createElement(component,props,...children)的语法糖，所有的JSX语法最终都会被转换成对这个方法的调用，虽然JSX只是一个语法糖，但使用它创建界面元素更加清晰简洁，在项目中应该首选JSX语法

**组件**

**组件定义**

**1**. 定义：组件将应用的UI拆分成独立的、可复用的模块，React应用程序正是由一个一个组件;

**2. 组件**两种方式：使用Es6 class(类组件)和使用函数(函数组件);

**3. class**定义组件：满足两个条件(1)class继承自React.Component;(2)class内部必须定义render方法，render方法返回代表该组件UI的React元素；

**组件的props**

1. 组件的props用于把父组件中的数据或方法传递给自组件，供子组件使用;

2. props是一个简单结构的对象，它包含的属性正是由组件作为JSX标签使用时的属性组成;

**组件的state**

1. 组件的state是组件内部的状态，state的变化最终将会反映到组件UI的变化上,在constructor中，通过this.state定义组件的状态，this.setState方法是改变组件状态的唯一方式，进而组件UI也会随之重新渲染;

2. 在组件的构造方法constructer内，首先要调用super(props)，这一步实际上是调用了React.Component这个class的constructor方法，用来完成React组件的初始化工作;

3. 组件的props和state都会直接影响组件的UI，事实上，React组件可以看作一个函数，函数的输入是props和state，函数的输出是组件的UI;props是对外的接口，组件通过props接收外部传入的数据(包括方法)、state是对内的接口，组件内部状态的变化通过state来反应，另外props是只读的，不能再组件内部修改props、state是可变的，组件状态的变化通过state来实现;

**有状态组件和无状态组件**

**1.** 定义：state用来反映组件内部状态的变化，如果一个组件的内部状态是不变的，当然就用不到state，这样的组件称之为无状态组件;反之，一个组件的内部状态发生变化，就需要使用state来保保存变化，这样的组件称之为有状态组件;

2. 定义无状态组件可以使用class的方式，还可以用函数定义，一个函数组件接收props作为参数，返回代表这个组件UI的React元素结构;

3. 有状态组件关注处理状态变化的业务逻辑，无状态组件主要关注组件UI的渲染;

4. 组件中解耦彻底才能更加容易被复用;

**属性校验和默认属性**

1. React提供了PropsTypes这个对象，用于校验组件属性的类型;

**组件样式**

**1. 外部CSS**样式表：使用组件的HTML页面中通过标签引入：;可以把样式表文件当作模块引入;

**2. 内**联样式：使用js对象表示css样式，驼峰式命名；style={{}},第一个大括号表示js表达式，第二个大括号表示js对象;

**组件和元素**

**1. Reac**t元素：React元素是一个普通的JavaScript的对象，这个对象可以通过DOM节点或React组件描述界面是什么样子的;

**2. Reac**t组件：React是一个class或者函数，它接收一些属性作为输入。返回一个React元素。React组件是由若干React元素组件而成的;

**组件的生命周期**

**定义**

组件从被创建到被销毁的过程称为组件的生命周期，React为组件在不同的生命周期阶段提供不同的生命周期方法

**挂载阶段**

这个阶段组件被创建，执行初始化，并挂在到DOM上，完成组件的第一次渲染，依次调用一下几个生命周期方法：

**1. construc**tor:它是ES6 class的构造方法，在这个方法中必须首先调用super(props)才能保证props被传入组件，constructor通常用于初始化组件的state以及绑定事件处理方法等工作;

**2. componentWillMo**unt:这个方法在组件被挂载到DOM前调用，且只会被调用一次，在这个组件中使用this.setState不会引起组件的重新渲染;

**3. ren**der:定义组件时唯一必要的方法(其他的方法可以省略)，它并不负责组件的实际渲染工作，它只是返回一个UI的描述，真正的渲染出页面DOM的工作由React自身负责。render是一个纯函数，在这个方法中不能执行任何有副作用的操作，所以不能在render中this.setState，这样会改变组件状态，会造成死循环;

**4. componentDidMo**unt:在组件挂载到DOM后调用，只调用一次，这个方法中通常还会用于像服务器端请求数据，在这个方法中调用this.setState会引起组件重新渲染;

**更新阶段**

组件被挂载到DOM后，组件的props或state可以引起组件的更新，依次调用的生命周期方法有:

**1. componentWillReceiveProps(nextPro**ps):这个组件只有在props引起的组件更新过程中，才会被调用，state引起组件的更新并不会触发该方法的执行，如果执行会进入死循环;

**2. shouldComponentUpdate(nextProps，nextSta**te):这个方法决定组件是否继续执行更新过程。当返回true时，组件会继续更新过程，反之返回为false时，组件的更新过程停止，后续的componentWillUpdate、render、componentDidUpdate也不会被调用，这样可以减少组件不必要的渲染，从而优化组件的性能,在该方法中不建议使用this.setState,会造成死循环问题，假如一定要使用，可以写判定条件，避免死循环;

**3. componentWillUpdate(nextProps，nextSta**te):可以作为组件更新发生前执行某些工作的地方，在该方法中不建议使用this.setState,会造成死循环问题，假如一定要使用，可以写判定条件，避免死循环;

**4. componentDidUpdate(prevProps，prevSta**te):prevProps，prevState代表组件更新前的props和state

**卸载阶段**

组件从DOM中被卸载的过程，只有一个生命周期方法:

**1. componentWillUNMo**unt:可以执行一些清理工作，比如清理组件中使用的定时器，清除componentDidMount中手动创建的DOM元素等，以避免内存泄漏 ;

**总结**

1. 只有类组件才具有生命周期方法，函数组件没有生命周期方法，所以永远不要在函数组件中使用生命周期方法

**列表和Keys**

1.React使用key属性来标记列表中每一个元素，当数据发生变化时，React就可以通过key知道哪些元素发生了变化，从而只重新渲染发生变化的元素，提高渲染效率。key不能重复，仅限于在一个列表中，不是所有列表中;

**事件处理**

React事件处理函数的写法主要有三种方式：

**1. 使用箭**头函数：直接在React元素中采用箭头函数定义事件的处理函数,箭头函数中的this指向的时函数定义时的对象，所以可以保证this总是指向当前组件的实例对象，直接在render方法中为元素事件定义事件处理函数，最大的问题是每次render调用时，都会重新创建一个新的事件处理函数，带来额外的性能开销;

**2. 使用组**件方法：直接将属性的方法赋值给元素的事件属性，同时在类的构造函数中，将这个方法的this绑定到当前对象，这种方式的好处是每次render不会重新创建一个回调函数，没有额外的性能损失；在为元素的事件属性赋值时，同时为事件处理函数绑定this，这种写法在render时都会创建一个新的函数，但是这种方式可以进行传递额外的参数;

**3. 属性初始**化语法：使用ES7的property initializers会自动的为class中定义的方法绑定this

**表单**

**1. 受**控组件：一个表单元素的值是由React来管理，那就是受控组件;

**2. 非**控组件：非控组件指表单元素的状态依然由表单元素自己管理，而不是交给React管理，使用非受控组件需要有一种方式可以获取到表单元素的值，React提供了一种特殊的属性ref，用来引用React组件或者DOM元素的实例;

**React16新特性**

**深入理解组件**

**组件state**

**设计合适的state**

1. state中的 所有状态都用于反映组件UI的变化，没有任何多余的状态，也不应该存在通过其他状态计算而来的中间状态;

2. state所代表的一个组件UI呈现的完整状态集又可以分成两类数据：用做渲染组件时用到的数据的来源以及用作组件UI展示形式的判断依据;

3. 除了state，props以外的其他组件属性称**为组件的**普通属性;

4. 组件中用到一个变量是不是应该作为state可以通过下面四条数据进行判断：

1. 这个变量是否通过props从父组件中获取？如果是，那么它不是一个状态;

2. 这个变量是否在组件的整个生命周期中都保持不变？如果是，那么它不是一个状态;

3. 这个变量是否可以通过其他状态(state)和属性(props)计算得到？如果是，那么它不是一个状态;

4. 这个变量是否在组件的render方法中使用？如果是，那么它不是一个状态，这种情况下，这个变量更适合定义为组件的一个普通属性;

**正确修改state**

**1. 不能直接修改st**ate：需要使用this.setState({})

**2. state的更新是**异步的：this.setState((preState,props)=>({}));

**3. state更新是一个合**并的过程

**state与不可变对象**

1.React推荐组件的状态是不可变对象： 不可变对象的修改会返回一个新对象，不需要担心原有对象在不小心的情况下被修改导致的错误，方便程序的管理和调试，在性能方面，在shouldComponentUpdate方法中仅需要比较前后两次状态对象的引用就可以判断状态是否真的改变，从而避免不必要的render

**组件与服务器通信**

**组件挂在阶段通信**

componentDidMount是执行组件与服务器通信最佳的地方，原因主要有两个：

1. 组件已经处于挂在状态，这是直接操作DOM是安全的，而componentWillMount无法保证这一点;

2. 当组件在在服务器端渲染时，componentWillMount会被调用两次，一次是在服务器端，另一次是在浏览器端，而componentDidMount只会执行一次，从而不会发生多余的数据请求;

**组件挂在阶段通信**

1. 在componentWillReceiveProps中进行更新时与服务端进行通信;

**组件通信**

**父子组件通信**

**兄弟组件通信**

**特殊的ref**

**DOM元素上使用ref**

**组件上使用ref**

**父组件访问子组件的DOM节点**

1. ref需要避免过度使用;

**虚拟DOM和性能优化**

**高阶组件**

**1. 基**本概念：高阶组件本质上是一个函数，这个函数接受一个组件作为输入，然后返回一个新的组件作为结果，而且，返回的新组件拥有了输入组件所不具备的功能;主要功能是封装并分离组件的通用逻辑，让通用逻辑在组件间更好的被复用。高阶组件的这种实现方式本质上是装饰者设计模式;

2. 根据返回的新组件和传入组件参数的关系，高阶组件的实现方式可以分为**两大类：代理方式的**高**阶组件和继承方式的**高阶组件;

**3. 代理方式的高**阶组件：高阶组件的特点是返回的新组件继承自React.Component类，新组件扮演的角色是传入参数组件的一个"代理",在新组件的render函数中，把被包裹组件渲染出来，除了高阶组件自己要做的工作，其余功能全部转手给被包裹**组件; 使**用场景：

**1. 操纵pr**ops:在被包装组件接收props前，高阶组件可以先拦截到props，对props执行增加、删除、修改的操作，然后将处理后的props在传给被包装的组件;

**2. 通过ref访问组**件实例:高阶组件通过ref获取被包装组件实例的引用，然后高阶组件就具备了直接操作被包装组件的属性或方法的能力;这个方法一般不建议使用;

**3. 组件状**态提升:高阶组件可以通过将包装组件的状态及相应的状态处理方法提升到高阶组件自身内部实现被包装组件的无状态化，应用场景:利用高阶组件将原本受控组件需要自己维护的状态统一提升到高阶组件中;

**4. 包**装组件:到目前为止，通过高阶组件产生的新组件，全部都是render函数直接返回被包裹组件，修改的只是props部分，其实render函数的JSX中完全可以引入其他元素，甚至可以组合多个React组件，这样可以对被包裹组件进行布局或样式的修改;

**4. 继承方式的高**阶组件:通过继承被包装的组件实现逻辑的复用，继承方式实现的高阶组件常用于渲染劫持、对被包装组件具有入侵性，当组合多个高阶组件使用时，很容易因为子类组件忘记通过super调用父类组件方法导致逻辑丢失，因此，在高阶组件中，应尽量使用代理**方式; 使**用场景：

1. 操纵props

2. 操纵生命周期函数

**5. 两种方**式区别：

1. 代理方式下render函数中的使用被包裹组件是通过JSX代码；继承方式下render函数中渲染被包裹组件的代码如下return super.render()

2. 代理方式下，被包裹组件经历一个完整的生命周期，继承方式下super.render只是一个生命周期中的一个函数而已;

3. 代理方式下产生的新组件和参数组件是两个不同的组件，一次渲染，两个组件都要经历各自的生命周期，继承方式下，只有一个生命周期,高阶组件中生命周期会覆盖被包裹组件的生命周期

**6. 高阶组件使**用方式：

**1. 使用正常函数进**行传参:const InputHoc = withFormHoc(WrappedComponent);

**2. 使**用插件:在react脚手架中运行npm eject脚本，能够看见所有安装的插件以及webpack配置文件，之后安装babel-preset-stage-2,babel-preset-react-native-stage-0,之后设置.babelrc中的内容如下:{"presets":["react-native-stage-0/decorator-support"]},之后使用高阶组件时可以直接写@withFormHoc,withFormHoc代表高阶组件名称，一般写在被包裹组件 定义的上面

**7. 高阶组件显**示名称:一般由高阶组件名字和被包裹组件的名字组成, static displayName = NewComponent${getDisplayName(WrappedComponent)} function getDisplayName(WrappedComponent){ return WrappedComponent.displayName ||WrappedComponent.name || 'Component' },这写在高阶组件里面，就可以让被包裹组件使用高阶组件时高阶组件有自己的名称

**8. 高阶组件使**用场景:利用高阶组件的条件渲染特性可以对页面进行权限控制，权限控制一般分为两个维度：页面级别和页面元素级别

**1. 页**面级别：
const withAuth = role => WrappedComponent => {
return class extends React.Component {
state = {
permission: false,
}
async componentWillMount() {
const currentRole = await getCurrentUserRole();
this.setState({
permission: currentRole === role,
});
}
render() {
if (this.state.permission) {
return ;
} else {
return (您没有权限查看该页面，请联系管理员！
);
}
}
};
}

**2. 页**面复用： . const withFetching = fetching => WrappedComponent => class extends React.Component {
state = {data: []}
async componentWillMount() {
const data = await fetching();
this.setState({data});
}
render() {
return ;
}
}
// pages/page-a.js
export default withFetching(fetching('science-fiction'))(MovieList);
// pages/page-b.js
export default withFetching(fetching('action'))(MovieList);
// pages/page-other.js
export default withFetching(fetching('some-other-type'))(MovieList);

**9. 与父组**件区别:从逻辑的执行流程上来看，高阶组件确实和父组件比较相像，但是高阶组件强调的是逻辑的抽象。高阶组件是一个函数，函数关注的是逻辑；父组件是一个组件，组件主要关注的是UI/DOM。如果逻辑是与DOM直接相关的，那么这部分逻辑适合放到父组件中实现；如果逻辑是与DOM不直接相关的，那么这部分逻辑适合使用高阶组件抽象，如数据校验、请求发送等。

**React-router**

**基本用法**

**1. 单页面应用和前**端路由：单页应用：它是一种网站应用的模型，它可以动态重写当前的页面来与用户交互，而不需要重新加载整个页面。单页应用的流畅性让 Web 应用更像桌面端或 Native 应用了。相对于传统的 Web 应用，单页应用做到了前后端分离，后端只负责处理数据提供接口，页面逻辑和页面渲染都交给了前端。前端发展到现在，单页应用的使用已经很广泛，目前时兴的 React、Vue、Angular 等前端框架均采用了 SPA 原则。单页应用意味着前端掌握了主动权，也让前端代码更复杂和庞大，模块化、组件化以及架构设计都变得越来越重要。原理：JS感知URL变化，当URL发生变化后，使用JS动态把当前的页面内容清除掉，再把下一个页面的内容挂载到页面上。此时的路由就不是后端来做了，而是前端来做，判断页面到底显示哪一个组件，再把以前的组件清除掉使用新的组件。就不会每一次跳转都请求HTML文件

**代码分片**

1. 运用路由进行代码分割，做到按需加载，提高应用加载速度

**2. asyncComponent&react-loada**ble：参考链接:https://www.jianshu.com/p/e00f9ea88c97