上课笔记

## 针对react的思考：

**声明式开发**：

只需要控制数据，不再直接去操作DOM节点；

以前：你盖一栋楼，要一步一步去做；

现在：你只需要提供图纸，剩下的事情React去完成。

**可以和其他框架并存**

React只负责管理指定的选择器下面的东西；

在此节点之外的节点，和React一点关系也没有，你可以引入其它框架来操作其它节点，相互之间是独立的；

**组件化开发：**

一个页面分隔成多个组件，组件之间可以相互传值；

**单项数据流：**

子组件只能使用值，不能修改父组件的值，如果修改值会报错，也就是props里面的数据只能使用不能修改；

修改数据还是父组件自己去完成的；好处是保证了数据的统一性；

**视图层框架：**

React只是视图层框架；

如果项目比较大还需要其他框架来支撑，比如组件之间传值比较多时需要使用Redux，单单靠react是不行的；AntDesign

**函数式编程：**

React代码里面都是一个个函数。

维护起来很容易，函数很大时可以进行拆分；

方案自动化测试等；

## PropTypes和DefaultProps：

PropTypes是定义prop的数据类型，用于限制传入的数据类型；

使用方法：

引入prop-types模块：架手架自带的模块，直接引入即可；

在组件类名上进行限制：

ShowMe.propTypes = {

showme:PropTypes.string,

age:PropTypes.oneOf([12,30,34]),

doing:PropTypes.oneOfType([PropTypes.string, PropTypes.number])

}

如果数据类型不满足要求，会发出警告，对开发非常友好。

如果一个属性没有传值，默认就不检测。

调用isRequired表示必须传值。

PropTypes.string.isRequired

DefaultProps：

用于设置默认值；

ShowMe.defaultProps = {

gender:'帅哥'

}

## Props，state和render的关系：

当组件的state或者props发生改变时，render函数就会重新执行。

修改子组件的数据不会影响父组件，父组件的render不会重新执行。

当父组件的render被运行时，他的子组件的render后会被重新执行。

## 什么是虚拟DOM

Render渲染的效率非常高，是因为render使用的是虚拟DOM概念。

那么虚拟DOM到底是什么那？

所谓的虚拟DOM就是用一个js对象来描述节点。

State 提供数据；

JSX 提供模板；

比如：

<div class=”artice”><p>你好，这是文章段落</p></div>

[‘div’, {class:’artice’}, [‘p’, {}, ‘你好，这是文章段落’]]

当state发生改变时，比较原始的虚拟DOM和新的虚拟DOM的差异；

找出差异后，直接对原始DOM进行操作，性能得到提升。

DIFF.

## CSS引入方式：

局部样式    命名规则: xxx.module.css

                  引入方式 import xxx from 'xxx.module.css'

                  用法：<div className={xxx.styleName}>

全局样式      命名规则: xxx.css

                    引入方式 import ‘xxx.css’

                    用法：<div className=*'styleName'*>

## 生命周期：

生命周期函数指在某一个时候自动调用执行的函数；

如：Render。

Constructor是js自带语法，不算作react的生命周期函数。

每个组件都有几个生命周期函数，以 will 为前缀的函数是在发生某些事之前调用，以 did 为前缀的是在发生某些事之后调用。

一个组件的生命周期过程：

**初始化过程**：initialization

初始化组件的props和state；

**挂载过程：**Mounting

组件即将挂载：componentWillMount，在组件即将被挂载到页面的时候执行。只会在即将挂载时执行，后面不会再执行。

把组件的内容渲染到页面：render

组件挂载完成：compontDidMount，组件挂载完成后执行，只在组件开始挂载并挂载完成后执行一次，后面不会再执行。

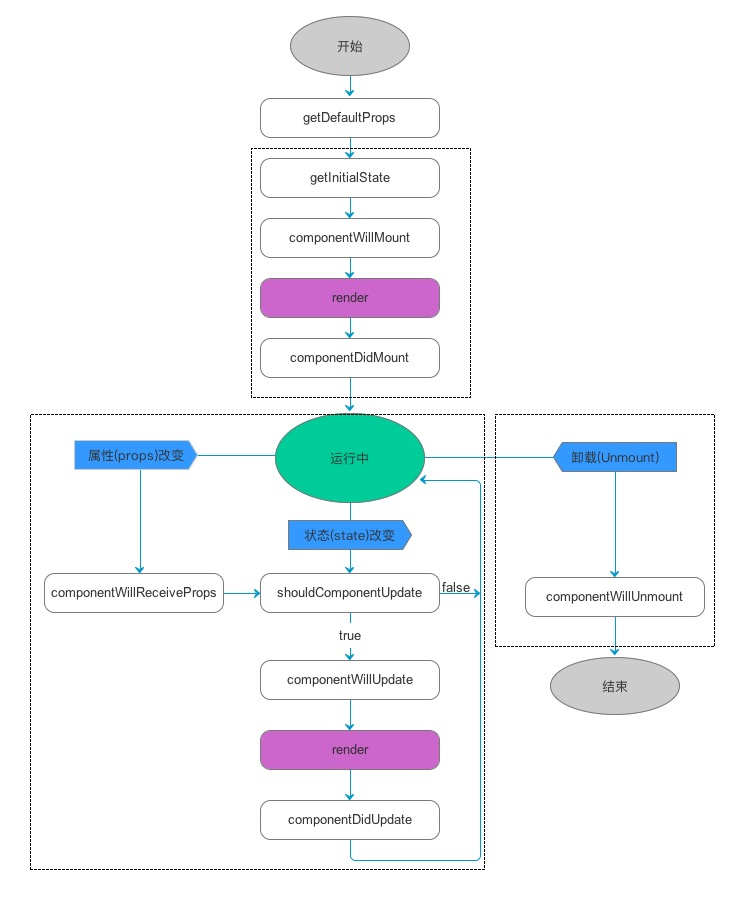
**更新过程：**updation

shouldComponentUpdate:确定组件是否更新，返回true或false，如果返回false后面的相关操作都不会执行；

componentWillUpdate：组件更新之前执行。

**组件卸载：**

componentWillUnmount：当组件即将被从页面中剔除的时候会被执行。



#### Mounting

如下这些方法在组件实例被创建和被插入到DOM中时被调用。

**constructor()：**

constructor在组件被mounted之前调用，我们的组件继承自React.Component，constructor函数中我们在其他操作前应该先调用 super(props)，否则this.props将会是undefined。

constructor 是初始化state的好地方。如果我们不需要初始化state，也不需要bind任何方法，那么在我们的组件中不需要实现constructor函数。

**componentWillMount()：**

此方法在mounting之前被立即调用，它在render()之前调用，因此在此方法中setState不会触发重新渲染。此方法是服务器渲染中调用的唯一的生命周期钩子，通常我们建议使用constructor()。

**render()：**render()方法是react组件必须的，它检查 this.props 和 this.state 并且返回一个React元素，我们也可以返回null或false，代表我们不想有任何的渲染。  
render()方法应该是一个纯方法，即它不会修改组件的state，在每一次调用时返回同样的结果。它不直接和浏览器交互，如果我们想要交互，应该在componentDidMount()或者其他的生命周期函数里面。

**componentDidMount()：**

此方法在组件被 mounted 之后立即被调用，初始化DOM节点应该在此方法中，如果需要从服务器端加载数据，这里是实例化网络请求的好地方，此方法中 setState 会触发组件重新渲染。

#### Updating

props 和 state 的改变产生更新。在重新渲染组件时，如下的方法被调用：

**componentWillReceiveProps()：**

一个已经mounted 的组件接收一个新的props之前 componentWillReceiveProps() 被调用，如果我们需要更新state来响应prop的更改，我们可以在此方法中比较 this.props 和 nextProps 并使用 this.setState 来更改state。

注意，即使props 没有改变，React也可以调用这个方法，因此如果你只想处理改变，请确保比较当前值和下一个值。当父组件导致你的组件重新渲染时，可能会发生这种情况：React在组件mounting 期间不会调用此方法，只有在一些组件的 props 可能被更新的时候才会调用。调用this.setState 通常不会触发componentWillReceiveProps。

**shouldComponentUpdate()：**

使用此方法让React 知道组件的输出是否不受当前state 或 props 更改的影响。默认行为是在每次state更改时重新渲染组件，在大多数情况下，我们应该默认改行为。

当接收到新的 props 或 state 时，shouldComponentUpdate() 在渲染之前被调用。默认返回true，对于初始渲染或使用 forceUpdate() 时，不调用此方法。返回false不会阻止子组件的state更改时，该子组件重新渲染。  
如果 shouldComponentUpdate() 返回false，那么 componentWillUpdate()，render() 和 componentDidUpdate() 将不会被调用。

**componentWillUpdate()：**

当接收新的 props 或 state 时，componentWillUpdate() 在组件渲染之前被立即调用。使用此函数作为在更新发生之前执行准备的机会。初始渲染不会调用此方法。  
注意：这里不能调用this.setState()  
如果调用会怎么样？试了一下，会产生死循环，一直更新。。。。如果我们需要更新state 以响应 props 的更改，我们应该使用componentWillReceiveProps()

**render()：**

render()方法是react组件必须的，它检查this.props 和 this.state 并且返回一个React元素，我们也可以返回null或false，代表我们不想有任何的渲染。  
render()方法应该是一个纯方法，即它不会修改组件的state，在每一次调用时返回同样的结果。它不直接和浏览器交互，如果我们想要交互，应该在componentDidMount() 或者其他的生命周期函数里面。

**componentDidUpdate()：**

此函数在更新后立即被调用。初始渲染不调用此方法。当组件已经更新时，使用此操作作为DOM操作的机会。这也是一个好的地方做网络请求，只要你比较当前的props 和以前的props (例如：如果props没有改变，可能不需要网络请求)。

#### Unmounting

当从dom中移除组件时，这个方法会被调用

**componentWillUnmount()：**

此函数在组件被卸载和销毁之前被立即调用。在此方法中执行一些必要的清理。例如清除计时器，取消网络请求或者清理在componentDidMount 中创建的任何DOM元素。

## 创建路由：

需要安装react-router-dom

引入：

import { BrowserRouter as Router, Route, Link } from 'react-router-dom';

创建router.js

import AllCom from './../components/Login'

import Calculator from './../components/Calculator'

import Welcome from './../components/Welcome'

const routes = [

{

path: '/',

component: AllCom,

exact: true

},

{

path: '/cal',

component: Calculator

},

{

path: '/wel',

component: Welcome

}

]

export default routes

在app.js里引入：

import routes from './model/router';

在需要连接的地方用link，在入口组件app里面匹配对应的组件。

return (

            <Router>

                <div>

                    <Link *to*="/">Home</Link> <br />

                    <Link *to*="/cal">User</Link><br />

                    <Link *to*='/wel'>News</Link>

                    {

                        routes.map((*v*, *i*) => {

                            if (v.exact) {

                                return <Route *key*={i} *exact* *path*={v.path} *component*={v.component} />

                            } else {

                                return <Route *key*={i} *path*={v.path} *component*={v.component} />

                            }

                        })

                    }

                </div>

            </Router>

        )

传参：

<https://www.jianshu.com/p/ad8cc02b9e6c>

<Route path='/web/departManange ' component={DepartManange}/>

<link to="web/departManange?tenantId=12">xxx</Link>

this.props.history.push({pathname:"/web/departManange?tenantId" + row.tenantId});

读取参数用: this.props.location.search

如果路由路径为:/path/:id

参数接收方式：this.props.match.params.id;

属性exact表示严格匹配，没有exact则模糊匹配,/path就会匹配/和/path两个路径，会显示两个路径对应的组件。

**函数式跳转：**

this.props.history.push('/list');

如果需要传值：

this.props.history.push({

pathname:'/list',

state:{

cid:101

}

});

接收值：

this.props.history.location

