**Webpack 基本配置**

## **入口(entry)**

****入口起点(entry point)****指示 webpack 应该使用哪个模块，来作为构建其内部*依赖图*的开始。进入入口起点后，webpack 会找出有哪些模块和库是入口起点（直接和间接）依赖的。

每个依赖项随即被处理，最后输出到称之为 *bundles* 的文件中，我们将在下一章节详细讨论这个过程。

可以通过在 [webpack 配置](https://www.webpackjs.com/configuration)中配置 entry 属性，来指定一个入口起点（或多个入口起点）。**默认值为 ./src**。 如果更换路径就要指定默认的js文件

接下来我们看一个 entry 配置的最简单例子：

****webpack.config.js****

module.exports = {

entry: './path/to/my/entry/file.js'};

}

## **出口(output)**

****output**** 属性告诉 webpack 在哪里输出它所创建的 *bundles*，以及如何命名这些文件，默认值为 ./dist。基本上，整个应用程序结构，都会被编译到你指定的输出路径的文件夹中。你可以通过在配置中指定一个 output 字段，来配置这些处理过程：

****webpack.config.js****

const path = require('path');

module.exports = {

entry: './path/to/my/entry/file.js',

output: {

path: path.resolve(\_\_dirname, 'dist'),

filename: 'my-first-webpack.bundle.js'

}};

## **loader**

*loader* 让 webpack 能够去处理那些非 JavaScript 文件（webpack 自身只理解 JavaScript）。loader 可以将所有类型的文件转换为 webpack 能够处理的有效[模块](https://www.webpackjs.com/concepts/modules)，然后你就可以利用 webpack 的打包能力，对它们进行处理。

本质上，webpack loader 将所有类型的文件，转换为应用程序的依赖图（和最终的 bundle）可以直接引用的模块。

*注意，loader 能够 import 导入任何类型的模块（例如 .css 文件），这是 webpack 特有的功能，其他打包程序或任务执行器的可能并不支持。我们认为这种语言扩展是有很必要的，因为这可以使开发人员创建出更准确的依赖关系图。*

在更高层面，在 webpack 的配置中 ****loader**** 有两个目标：

1. test 属性，用于标识出应该被对应的 loader 进行转换的某个或某些文件。
2. use 属性，表示进行转换时，应该使用哪个 loader。

****webpack.config.js****

const path = require('path');

const config = {

output: {

filename: 'my-first-webpack.bundle.js'

},

module: {

rules: [

{ test: /\.txt$/, use: 'raw-loader' }

]

}};

module.exports = config;

以上配置中，对一个单独的 module 对象定义了 rules 属性，里面包含两个必须属性：test 和 use。这告诉 webpack 编译器(compiler) 如下信息：

“嘿，webpack 编译器，当你碰到「在 require()/import 语句中被解析为 '.txt' 的路径」时，在你对它打包之前，先****使用**** raw-loader 转换一下。”

 module: {

      strictExportPresence: true,

      rules: [

        // Disable require.ensure as it's not a standard language feature.

        { parser: { requireEnsure: false } },

        // First, run the linter.

        // It's important to do this before Babel processes the JS.

// jsLoader

        {

          test: /\.(js|mjs|jsx|ts|tsx)$/,

          enforce: 'pre',

          use: [

            {

              options: {

                cache: true,

// 格式化

                formatter: require.resolve('react-dev-utils/eslintFormatter'),

                eslintPath: require.resolve('eslint'),

                resolvePluginsRelativeTo: \_\_dirname,

              },

              loader: require.resolve('eslint-loader'),

            },

          ],

          include: paths.appSrc,

        },

        {

          // "oneOf" will traverse all following loaders until one will

          // match the requirements. When no loader matches it will fall

          // back to the "file" loader at the end of the loader list.

          oneOf: [

            // "url" loader works like "file" loader except that it embeds assets

            // smaller than specified limit in bytes as data URLs to avoid requests.

            // A missing `test` is equivalent to a match.

// img loader

            {

              test: [/\.bmp$/, /\.gif$/, /\.jpe?g$/, /\.png$/],

              loader: require.resolve('url-loader'),

              options: {

// 当图片的大小超过 imageInlineSizeLimit 时采用 imgLoader

// 默认吧图片处理成base64

                limit: imageInlineSizeLimit,

                name: 'static/media/[name].[hash:8].[ext]',

              },

            },

            // Process application JS with Babel.

            // The preset includes JSX, Flow, TypeScript, and some ESnext features.

// 用babel 处理js 文件

            {

              test: /\.(js|mjs|jsx|ts|tsx)$/,

              include: paths.appSrc,

              loader: require.resolve('babel-loader'),

              options: {

                customize: require.resolve(

                  'babel-preset-react-app/webpack-overrides'

                ),

                plugins: [

                  [

                    require.resolve('babel-plugin-named-asset-import'),

                    {

                      loaderMap: {

                        svg: {

                          ReactComponent:

                            '@svgr/webpack?-svgo,+titleProp,+ref![path]',

                        },

                      },

                    },

                  ],

                ],

                // This is a feature of `babel-loader` for webpack (not Babel itself).

                // It enables caching results in ./node\_modules/.cache/babel-loader/

                // directory for faster rebuilds.

                cacheDirectory: true,

                // See #6846 for context on why cacheCompression is disabled

                cacheCompression: false,

                compact: isEnvProduction,

              },

            },

            // Process any JS outside of the app with Babel.

            // Unlike the application JS, we only compile the standard ES features.

            {

              test: /\.(js|mjs)$/,

              exclude: /@babel(?:\/|\\{1,2})runtime/,

              loader: require.resolve('babel-loader'),

              options: {

                babelrc: false,

                configFile: false,

                compact: false,

                presets: [

                  [

                    require.resolve('babel-preset-react-app/dependencies'),

                    { helpers: true },

                  ],

                ],

                cacheDirectory: true,

                // See #6846 for context on why cacheCompression is disabled

                cacheCompression: false,

                // Babel sourcemaps are needed for debugging into node\_modules

                // code.  Without the options below, debuggers like VSCode

                // show incorrect code and set breakpoints on the wrong lines.

                sourceMaps: shouldUseSourceMap,

                inputSourceMap: shouldUseSourceMap,

              },

            },

            // "postcss" loader applies autoprefixer to our CSS.

            // "css" loader resolves paths in CSS and adds assets as dependencies.

            // "style" loader turns CSS into JS modules that inject <style> tags.

            // In production, we use MiniCSSExtractPlugin to extract that CSS

            // to a file, but in development "style" loader enables hot editing

            // of CSS.

            // By default we support CSS Modules with the extension .module.css

            {

              test: cssRegex,

              exclude: cssModuleRegex,

              use: getStyleLoaders({

                importLoaders: 1,

                sourceMap: isEnvProduction && shouldUseSourceMap,

              }),

              // Don't consider CSS imports dead code even if the

              // containing package claims to have no side effects.

              // Remove this when webpack adds a warning or an error for this.

              // See https://github.com/webpack/webpack/issues/6571

              sideEffects: true,

            },

            // Adds support for CSS Modules (https://github.com/css-modules/css-modules)

            // using the extension .module.css

            {

              test: cssModuleRegex,

              use: getStyleLoaders({

                importLoaders: 1,

                sourceMap: isEnvProduction && shouldUseSourceMap,

                modules: {

                  getLocalIdent: getCSSModuleLocalIdent,

                },

              }),

            },

            // Opt-in support for SASS (using .scss or .sass extensions).

            // By default we support SASS Modules with the

            // extensions .module.scss or .module.sass

// sass loader

            {

              test: sassRegex,

              exclude: sassModuleRegex,

              use: getStyleLoaders(

                {

                  importLoaders: 3,

                  sourceMap: isEnvProduction && shouldUseSourceMap,

                },

                'sass-loader'

              ),

              // Don't consider CSS imports dead code even if the

              // containing package claims to have no side effects.

              // Remove this when webpack adds a warning or an error for this.

              // See https://github.com/webpack/webpack/issues/6571

              sideEffects: true,

            },

            // Adds support for CSS Modules, but using SASS

            // using the extension .module.scss or .module.sass

            {

              test: sassModuleRegex,

              use: getStyleLoaders(

                {

                  importLoaders: 3,

                  sourceMap: isEnvProduction && shouldUseSourceMap,

                  modules: {

                    getLocalIdent: getCSSModuleLocalIdent,

                  },

                },

                'sass-loader'

              ),

            },

            // "file" loader makes sure those assets get served by WebpackDevServer.

            // When you `import` an asset, you get its (virtual) filename.

            // In production, they would get copied to the `build` folder.

            // This loader doesn't use a "test" so it will catch all modules

            // that fall through the other loaders.

// 文件loader

            {

              loader: require.resolve('file-loader'),

              // Exclude `js` files to keep "css" loader working as it injects

              // its runtime that would otherwise be processed through "file" loader.

              // Also exclude `html` and `json` extensions so they get processed

              // by webpacks internal loaders.

              exclude: [/\.(js|mjs|jsx|ts|tsx)$/, /\.html$/, /\.json$/],

              options: {

                name: 'static/media/[name].[hash:8].[ext]',

              },

            },

            // \*\* STOP \*\* Are you adding a new loader?

            // Make sure to add the new loader(s) before the "file" loader.

          ],

        },

      ],

    },

## **插件(plugins)**

loader 被用于转换某些类型的模块，而插件则可以用于执行范围更广的任务。插件的范围包括，从打包优化和压缩，一直到重新定义环境中的变量。[插件接口](https://www.webpackjs.com/api/plugins)功能极其强大，可以用来处理各种各样的任务。

想要使用一个插件，你只需要 require() 它，然后把它添加到 plugins 数组中。多数插件可以通过选项(option)自定义。你也可以在一个配置文件中因为不同目的而多次使用同一个插件，这时需要通过使用 new 操作符来创建它的一个实例。

****webpack.config.js****

const HtmlWebpackPlugin = require('html-webpack-plugin'); // 通过 npm 安装const webpack = require('webpack'); // 用于访问内置插件

const config = {

module: {

rules: [

{ test: /\.txt$/, use: 'raw-loader' }

]

},

plugins: [

new HtmlWebpackPlugin({template: './src/index.html'})

]};

module.exports = config;

webpack 提供许多开箱可用的插件！查阅我们的[插件列表](https://www.webpackjs.com/plugins)获取更多信息。

在 webpack 配置中使用插件是简单直接的，然而也有很多值得我们进一步探讨的用例。

[了解更多！](https://www.webpackjs.com/concepts/plugins)

plugins: [

      // Generates an `index.html` file with the <script> injected.

      new HtmlWebpackPlugin(

        Object.assign(

          {},

          {

            inject: true,

            template: paths.appHtml,

          },

          isEnvProduction

            ? {

                minify: {

                  removeComments: true,

                  collapseWhitespace: true,

                  removeRedundantAttributes: true,

                  useShortDoctype: true,

                  removeEmptyAttributes: true,

                  removeStyleLinkTypeAttributes: true,

                  keepClosingSlash: true,

                  minifyJS: true,

                  minifyCSS: true,

                  minifyURLs: true,

                },

              }

            : undefined

        )

      ),

      // Inlines the webpack runtime script. This script is too small to warrant

      // a network request.

      // https://github.com/facebook/create-react-app/issues/5358

      isEnvProduction &&

        shouldInlineRuntimeChunk &&

        new InlineChunkHtmlPlugin(HtmlWebpackPlugin, [/runtime-.+[.]js/]),

      // Makes some environment variables available in index.html.

      // The public URL is available as %PUBLIC\_URL% in index.html, e.g.:

      // <link rel="icon" href="%PUBLIC\_URL%/favicon.ico">

      // In production, it will be an empty string unless you specify "homepage"

      // in `package.json`, in which case it will be the pathname of that URL.

      // In development, this will be an empty string.

      new InterpolateHtmlPlugin(HtmlWebpackPlugin, env.raw),

      // This gives some necessary context to module not found errors, such as

      // the requesting resource.

      new ModuleNotFoundPlugin(paths.appPath),

      // Makes some environment variables available to the JS code, for example:

      // if (process.env.NODE\_ENV === 'production') { ... }. See `./env.js`.

      // It is absolutely essential that NODE\_ENV is set to production

      // during a production build.

      // Otherwise React will be compiled in the very slow development mode.

      new webpack.DefinePlugin(env.stringified),

      // This is necessary to emit hot updates (currently CSS only):

      isEnvDevelopment && new webpack.HotModuleReplacementPlugin(),

      // Watcher doesn't work well if you mistype casing in a path so we use

      // a plugin that prints an error when you attempt to do this.

      // See https://github.com/facebook/create-react-app/issues/240

      isEnvDevelopment && new CaseSensitivePathsPlugin(),

      // If you require a missing module and then `npm install` it, you still have

      // to restart the development server for Webpack to discover it. This plugin

      // makes the discovery automatic so you don't have to restart.

      // See https://github.com/facebook/create-react-app/issues/186

      isEnvDevelopment &&

        new WatchMissingNodeModulesPlugin(paths.appNodeModules),

      isEnvProduction &&

        new MiniCssExtractPlugin({

          // Options similar to the same options in webpackOptions.output

          // both options are optional

          filename: 'static/css/[name].[contenthash:8].css',

          chunkFilename: 'static/css/[name].[contenthash:8].chunk.css',

        }),

      // Generate an asset manifest file with the following content:

      // - "files" key: Mapping of all asset filenames to their corresponding

      //   output file so that tools can pick it up without having to parse

      //   `index.html`

      // - "entrypoints" key: Array of files which are included in `index.html`,

      //   can be used to reconstruct the HTML if necessary

      new ManifestPlugin({

        fileName: 'asset-manifest.json',

        publicPath: publicPath,

        generate: (seed, files, entrypoints) => {

          const manifestFiles = files.reduce((manifest, file) => {

            manifest[file.name] = file.path;

            return manifest;

          }, seed);

          const entrypointFiles = entrypoints.main.filter(

            fileName => !fileName.endsWith('.map')

          );

          return {

            files: manifestFiles,

            entrypoints: entrypointFiles,

          };

        },

      }),

      // Moment.js is an extremely popular library that bundles large locale files

      // by default due to how Webpack interprets its code. This is a practical

      // solution that requires the user to opt into importing specific locales.

      // https://github.com/jmblog/how-to-optimize-momentjs-with-webpack

      // You can remove this if you don't use Moment.js:

      new webpack.IgnorePlugin(/^\.\/locale$/, /moment$/),

      // Generate a service worker script that will precache, and keep up to date,

      // the HTML & assets that are part of the Webpack build.

      isEnvProduction &&

        new WorkboxWebpackPlugin.GenerateSW({

          clientsClaim: true,

          exclude: [/\.map$/, /asset-manifest\.json$/],

          importWorkboxFrom: 'cdn',

          navigateFallback: publicUrl + '/index.html',

          navigateFallbackBlacklist: [

            // Exclude URLs starting with /\_, as they're likely an API call

            new RegExp('^/\_'),

            // Exclude any URLs whose last part seems to be a file extension

            // as they're likely a resource and not a SPA route.

            // URLs containing a "?" character won't be blacklisted as they're likely

            // a route with query params (e.g. auth callbacks).

            new RegExp('/[^/?]+\\.[^/]+$'),

          ],

        }),

      // TypeScript type checking

      useTypeScript &&

        new ForkTsCheckerWebpackPlugin({

          typescript: resolve.sync('typescript', {

            basedir: paths.appNodeModules,

          }),

          async: isEnvDevelopment,

          useTypescriptIncrementalApi: true,

          checkSyntacticErrors: true,

          resolveModuleNameModule: process.versions.pnp

            ? `${\_\_dirname}/pnpTs.js`

            : undefined,

          resolveTypeReferenceDirectiveModule: process.versions.pnp

            ? `${\_\_dirname}/pnpTs.js`

            : undefined,

          tsconfig: paths.appTsConfig,

          reportFiles: [

            '\*\*',

            '!\*\*/\_\_tests\_\_/\*\*',

            '!\*\*/?(\*.)(spec|test).\*',

            '!\*\*/src/setupProxy.\*',

            '!\*\*/src/setupTests.\*',

          ],

          silent: true,

          // The formatter is invoked directly in WebpackDevServerUtils during development

          formatter: isEnvProduction ? typescriptFormatter : undefined,

        }),

    ].filter(Boolean),

    // Some libraries import Node modules but don't use them in the browser.

    // Tell Webpack to provide empty mocks for them so importing them works.

    node: {

      module: 'empty',

      dgram: 'empty',

      dns: 'mock',

      fs: 'empty',

      http2: 'empty',

      net: 'empty',

      tls: 'empty',

      child\_process: 'empty',

    },

    // Turn off performance processing because we utilize

    // our own hints via the FileSizeReporter

    performance: false,

  };

React 学习

.执行 npm run eject 将显示所有的配置文件，但是不可返回

<https://www.webpackjs.com/loaders/> 配置loader 地址

<https://reactjs.bootcss.com/> 中文文档官网

只要在jsx 里编写javascript 代码就加一层 {} 括号

React 组件有函数创建和类创建两种方式

函数创建

function App() {

  return (

    <div className="App">

       this is my react

    </div>

  );

}

类创建 通过props 来进父子间参数的传递

class NewAPP extends React.Component {

    render() {

        return (

            <div>

                this class react

                <h1>{this.props.desc}</h1>

            </div>

        )

    }

}

类创建的原理:

const app = new NewAPP({

    desc: "react 组件化的开发"

}).render()

JSX 实现原理

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

代码样列:

import React from 'react';

import ReactDOM from 'react-dom';

import './index.css';

import App from './App';

import \* as serviceWorker from './serviceWorker';

class NewAPP extends React.Component {

    render() {

        return (

            <div class="Root">

                <h1>JSX实现原理</h1>

                <h1>利用react来创建</h1>

            </div>

        )

    }

}

class JsxApp extends React.Component {

     /\* JSX 本来不是javaScript 代码   利用 React.createElement 来编译创建元素\*/

    render() {

        return (

            React.createElement(

                'div',

                // 属性

                {

                   className: 'Root'

                },

                 // 子节点

                React.createElement(

                    'h1',

                    {

                        title: 'JSX实现原理'

                    },

                    'JSX实现原理'

                ),

                React.createElement('h1',

                {

                    title: '利用react来创建'

                },

                '利用react来创建'

                )

            )

        )

    }

}

ReactDOM.render(<JsxApp />, document.getElementById('root'));

// If you want your app to work offline and load faster, you can change

// unregister() to register() below. Note this comes with some pitfalls.

// Learn more about service workers: https://bit.ly/CRA-PWA

serviceWorker.unregister();

React 的类及样式

Npm 官网 [https://www.npmjs.com](https://www.npmjs.com/) 以后有什么npm 相关的问题可以上这个官网进行搜索

npm install classnames --save 安装classnames 可以对元素的类名进行管理

import classNames from 'classnames'

render() {

        return(

            <div>

                <h1 className={classNames('a','b')}>this is Title</h1>

            </div>

        )

}

可以给一个元素添加多个className 值

如果不想要渲染时默认的父容器标签 可以用react 自带的fragement 标签 或者<></> 空标签进行过滤

代码示例:

import React, { Fragment } from 'react';

class TodoHeader extends React.Component {

    render() {

        return(

         /\*  <Fragment>

               TodoHeader

            </Fragment>  \*/

            <>

            TodoHeader

            </>

        )

    }

}

export default TodoHeader

导出管理文件的方式有两种 但上面一种比较好

示例:

export { default as TodoHeader } from './todoHeader'

export { default as TodoInput } from './todoInput'

export { default as TodoList } from './todoList'

// 或者另一种导出方式

/\*

import TodoHeader from './todoHeader'

import TodoInput from './TodoInput'

import TodoList from './TodoList'

export {

    TodoHeader,

    TodoInput,

    TodoList

} \*/

React 父子间通信 props

****所有 React 组件都必须像纯函数一样保护它们的 props 不被更改。****

可以安装 propTypes 来进行类型检测

npm install --save prop-types

import React from 'react';

import { TodoHeader, TodoList, TodoInput } from './components'

function App() {

  return (

    <div className="App">

       <TodoHeader desc="待办事项列表"><i>这里是待办事项</i></TodoHeader>

       <TodoInput />

       <TodoList />

    </div>

  );

}

export default App;

组件式的定义

import PropTypes from 'prop-types'

class TodoInput extends React.Component {

    static propTypes = {

       desc: PropTypes.string

    }

    static defaultProps = {

        desc: '添加'

    }

    render() {

        return(

            <div>

               <input type="text" />

               <button>{this.props.desc}</button>

            </div>

        )

    }

}

export default TodoInput

函数式的定义

import React from 'react';

import PropTypes  from 'prop-types'

export default function TodoHeader(props) {

    return (

        <>

        <h1>{props.desc}</h1>

        <h1>{props.x + props.y}</h1>

        {props.children}

        </>

    )

}

TodoHeader.propTypes = {

    desc: PropTypes.string.isRequired,

    x: PropTypes.number.isRequired,

    y: PropTypes.number.isRequired

}

TodoHeader.defaultProps = {

    x: 4,

    y: 7

}

可以使用 dangerouslySetInnerHTML 来实现渲染标签元素，但是这样会引发xss注入问题

 <div  dangerouslySetInnerHTML={{\_\_html: this.state.temp}}></div>

State 状态值

State 的申明方式有两种 一种是在constructor 中用this.state 申明

一种是直接在组件中直接用state 关键字来申明 **注意：** 只有类组件有自己的state 值，函数组件没有自己的state 值

 constructor() {

    super()

    this.state = {

      todoList: [

        {

          id: 1,

          title: '吃饭',

          isCompleted: true

        },

        {

          id: 2,

          title: '睡觉',

          isCompleted: false

        }

      ],

    }

  }

改变state 状态值的方式有两种 均在代码中有注释

代码样列

import React from 'react';

class Liked extends React.Component {

    state = {

        isLiked: false

    }

    handleClick = () => {

      // 第一中改变state 的方式是直接传一个对象的形式

    /\*  this.setState({

          isLiked: !this.state.isLiked

      }) \*/

      // 改变state 的第二种方式是通过函数形式来改变  有两个参数可以传递  第一个表示上一次的state 值 第二个表示

      // 上一次的props 值

      // 由于setState 是异步的，所以要想获取到state 改变后的状态值就要传递第二个函数参数

      this.setState((preState, props) => {

        console.log("this is inner state", this.state.isLiked)

           return {

               isLiked: !preState.isLiked

           }

      }, () => {

          console.log(this.state.isLiked)

      })

      console.log("this is outer state", this.state.isLiked)

    }

    render() {

        console.log(this.state)

        return(

           <div>

             <h1 onClick = { this.handleClick }>{this.state.isLiked ? '喜欢' : '不喜欢'} </h1>

           </div>

        )

    }

}

export default Liked

React 事件处理

1. Input 框要绑定onChange 事件之后才能进行输入

handleChange = (e, id) => {

      //  console.log(e.currentTarget, id)

        this.setState({

            inputValue: e.currentTarget.value

        })

    }

 <input type="text" value={ this.state.inputValue } onChange={ this.handleChange.bind(this) }/>

1. onClick 事件

有两种绑定事件的方法 一种是直接在constructor 中使用bind 绑定

一种是直接在父组件中将事件传递给子组件

// 绑定this 的方式有两种 第一种直接在onChange 事件上绑定 this.handleChange.bind(this); 但这种方法每次

    // 刷新render 就会调用一次   会生产多个绑定事件  不好

    // 第二种比较好的是在constructor 中直接绑定this.handleChange = this.handleChange.bind(this);

    // 如果要对事件进行传参  那么使用bind 绑定的方式可以

    // 但是 react 官方推荐的是在父组件中将方法传递给子组件  然后由子组件去触发方法

    // 这种绑定传参的方式不推荐    <button onClick = { () => {this.handleClick(this.state.inputValue)} }>{this.props.desc}</button>

handleTodo= (value)=> {

  // console.log(...this.state.TodoList)

  this.setState({

    todoList: this.state.todoList.concat({

      id: Math.random(),

      title: value,

      isCompleted: false

    })

  })

 }

  render() {

    return (

      <div className="App">

         <TodoHeader desc="待办事项列表"></TodoHeader>

         <TodoInput handleTodo = { this.handleTodo }/>

 constructor() {

        super()

        this.state = {

            inputValue: 'xxx'

        }

        this.handleChange = this.handleChange.bind(this);

    }

    handleChange = (e, id) => {

      //  console.log(e.currentTarget, id)

        this.setState({

            inputValue: e.currentTarget.value

        })

    }

    handleClick = (id) => {

        this.props.handleTodo(this.state.inputValue)

    }

    // 绑定this 的方式有两种 第一种直接在onChange 事件上绑定 this.handleChange.bind(this); 但这种方法每次

    // 刷新render 就会调用一次   会生产多个绑定事件  不好

    // 第二种比较好的是在constructor 中直接绑定this.handleChange = this.handleChange.bind(this);

    // 如果要对事件进行传参  那么使用bind 绑定的方式可以

    // 但是 react 官方推荐的是在父组件中将方法传递给子组件  然后由子组件去触发方法

    // 这种绑定传参的方式不推荐    <button onClick = { () => {this.handleClick(this.state.inputValue)} }>{this.props.desc}</button>

    render() {

        console.log(this.props)

        return(

            <div>

               <input type="text" value={ this.state.inputValue } onChange={ this.handleChange.bind(this) }/>

               <button onClick = { this.handleClick }>{this.props.desc}</button>

            </div>

        )

    }

}

### 表单事件

### 事件名：

### onChange onInput onInvalid onSubmit

表单事件的事件类型有很多，具体要参考官方文档去做处理

这里举出了一些列子

return(

            <div>

               <input

               type="text"

               ref = { this.inputDom }

               value={ this.state.inputValue }

               onChange={ this.handleChange.bind(this) }

               onKeyUp = {this.handleInputKeyUp}

               />

               <button

               onClick = { this.handleClick }

               onPointerEnter = {this.onPointerEnterEvent}

               >

               {this.props.desc}

               </button>

            </div>

        )

    }

 handleInputKeyUp = (e) => {

       if(e.keyCode === 13 ){

           this.handleClick();

       }

    }

    handleClick = () => {

        this.props.handleTodo(this.state.inputValue)

        this.setState({

                inputValue: ''

            },() => {

           this.inputDom.current.focus()

        })

    }

    onPointerEnterEvent = () => {

        console.log("onPointerEnter")

    }

对事件的处理方法都类似

Ref 标签

### 创建 Refs

Refs 是使用 React.createRef() 创建的，并通过 ref 属性附加到 React 元素。在构造组件时，通常将 Refs 分配给实例属性，以便可以在整个组件中引用它们。

class MyComponent extends React.Component {

constructor(props) {

super(props); this.myRef = React.createRef(); }

render() { return <div ref={this.myRef} />; }}

### 访问 Refs

当 ref 被传递给 render 中的元素时，对该节点的引用可以在 ref 的 current 属性中被访问。

const node = this.myRef.current;

ref 的值根据节点的类型而有所不同：

* 当 ref 属性用于 HTML 元素时，构造函数中使用 React.createRef() 创建的 ref 接收底层 DOM 元素作为其 current 属性。
* 当 ref 属性用于自定义 class 组件时，ref 对象接收组件的挂载实例作为其 current 属性。
* ****你不能在函数组件上使用**ref**属性****，因为他们没有实例。

例子:

import React, { createRef } from 'react';

 this.inputDom = createRef()

 <input

               type="text"

               ref = { this.inputDom }

 handleClick = () => {

        this.props.handleTodo(this.state.inputValue)

        this.setState({

                inputValue: ''

            },() => {

           this.inputDom.current.focus()

        })

    }

简而言之就是讲一个元素挂在到这个inputDom 上去

组件的生命周期

重点:

函数没有自己的生命周期，只有组件有自己的生命周期

只有setState 或者props 发生改变, forceUpdate 调用的时候组件才会更新，生命周期才会相应的调用

static getDerivedStateFromProps()

注意这个方法，因为它是静态方法，所以它没有this 指针

PureComponent

当使用继承这个组件时数据更新的时候会进行一次浅比较，对于数据没有改变时不去调用render 方法

PureComponent 会对 props 和 state 进行浅层比较，并减少了跳过必要更新的可能性。

当 props 或 state 发生变化时，shouldComponentUpdate() 会在渲染执行之前被调用。返回值默认为 true。首次渲染或使用 forceUpdate() 时不会调用该方法。

此方法仅作为**[性能优化的方式](https://zh-hans.reactjs.org/docs/optimizing-performance.html)**而存在。不要企图依靠此方法来“阻止”渲染，因为这可能会产生 bug。你应该****考虑使用内置的**[PureComponent](https://zh-hans.reactjs.org/docs/react-api.html" \l "reactpurecomponent)**组件****，而不是手动编写 shouldComponentUpdate()。PureComponent 会对 props 和 state 进行浅层比较，并减少了跳过必要更新的可能性。

如果你一定要手动编写此函数，可以将 this.props 与 nextProps 以及 this.state 与nextState 进行比较，并返回 false 以告知 React 可以跳过更新。请注意，返回 false 并不会阻止子组件在 state 更改时重新渲染。

我们不建议在 shouldComponentUpdate() 中进行深层比较或使用 JSON.stringify()。这样非常影响效率，且会损害性能。

目前，如果 shouldComponentUpdate() 返回 false，则不会调用 [UNSAFE\_componentWillUpdate()](https://zh-hans.reactjs.org/docs/react-component.html" \l "unsafe_componentwillupdate)，[render()](https://zh-hans.reactjs.org/docs/react-component.html" \l "render) 和 [componentDidUpdate()](https://zh-hans.reactjs.org/docs/react-component.html" \l "componentdidupdate)。后续版本，React 可能会将 shouldComponentUpdate 视为提示而不是严格的指令，并且，当返回 false 时，仍可能导致组件重新渲染。

虽然shouldComponentUpdate可以对state 进行深度过滤 但是不推荐使用，会严重影响效率

例子:

  shouldComponentUpdate(nextProps, nextState){

      //  console.log(nextProps)

        return nextProps.isCompleted === this.props.isCompleted

    }

如果涉及多层json判断介意使用 loadsh来进行比较 同样也特别影响效率

npm install react-lodash

import  \_  from 'lodash'

 shouldComponentUpdate(nextProps, nextState){

      //  console.log(nextProps)

      return \_.isEqual(nextProps, this.props)

      //  return nextProps.isCompleted === this.props.isCompleted

    }

### 组件的生命周期

每个组件都包含“生命周期方法”，你可以重写这些方法，以便于在运行过程中特定的阶段执行这些方法。****你可以使用此**[生命周期图谱](http://projects.wojtekmaj.pl/react-lifecycle-methods-diagram/" \t "https://zh-hans.reactjs.org/docs/_blank)**作为速查表****。在下述列表中，常用的生命周期方法会被加粗。其余生命周期函数的使用则相对罕见。

#### 挂载

当组件实例被创建并插入 DOM 中时，其生命周期调用顺序如下：

* **[constructor()](https://zh-hans.reactjs.org/docs/react-component.html" \l "constructor)**
* [static getDerivedStateFromProps()](https://zh-hans.reactjs.org/docs/react-component.html" \l "static-getderivedstatefromprops)
* **[render()](https://zh-hans.reactjs.org/docs/react-component.html" \l "render)**
* **[componentDidMount()](https://zh-hans.reactjs.org/docs/react-component.html" \l "componentdidmount)**

**注意:**

下述生命周期方法即将过时，在新代码中应该[避免使用它们](https://zh-hans.reactjs.org/blog/2018/03/27/update-on-async-rendering.html)：

* [UNSAFE\_componentWillMount()](https://zh-hans.reactjs.org/docs/react-component.html" \l "unsafe_componentwillmount)

#### 更新

当组件的 props 或 state 发生变化时会触发更新。组件更新的生命周期调用顺序如下：

* [static getDerivedStateFromProps()](https://zh-hans.reactjs.org/docs/react-component.html" \l "static-getderivedstatefromprops)
* [shouldComponentUpdate()](https://zh-hans.reactjs.org/docs/react-component.html" \l "shouldcomponentupdate)
* **[render()](https://zh-hans.reactjs.org/docs/react-component.html" \l "render)**
* [getSnapshotBeforeUpdate()](https://zh-hans.reactjs.org/docs/react-component.html" \l "getsnapshotbeforeupdate)
* **[componentDidUpdate()](https://zh-hans.reactjs.org/docs/react-component.html" \l "componentdidupdate)**

**注意:**

下述方法即将过时，在新代码中应该[避免使用它们](https://zh-hans.reactjs.org/blog/2018/03/27/update-on-async-rendering.html)：

* [UNSAFE\_componentWillUpdate()](https://zh-hans.reactjs.org/docs/react-component.html" \l "unsafe_componentwillupdate)
* [UNSAFE\_componentWillReceiveProps()](https://zh-hans.reactjs.org/docs/react-component.html" \l "unsafe_componentwillreceiveprops)

#### 卸载

当组件从 DOM 中移除时会调用如下方法：

* **[componentWillUnmount()](https://zh-hans.reactjs.org/docs/react-component.html" \l "componentwillunmount)**

#### 错误处理

当渲染过程，生命周期，或子组件的构造函数中抛出错误时，会调用如下方法：

* [static getDerivedStateFromError()](https://zh-hans.reactjs.org/docs/react-component.html" \l "static-getderivedstatefromerror)
* [componentDidCatch()](https://zh-hans.reactjs.org/docs/react-component.html" \l "componentdidcatch)

### Hook

### 是react 提供的专门为了函数组件提供state 和生命周期的函数

其中 useState 相当于 class 组件的state 和setState. useEffect 相当于 class 组件的componentDidMount 和componentDidUpdate

代码示例:

const Counter = () => {

    // useState 相当于 class 的state 和 setState()

    const [count, setCount] = useState(0)

     // 相当于class 的comoponentDidMount 和 componentDidUpdate

    useEffect(() => {

        document.title = `你点击了${count}次`

    })

    return (

        <div>

          <button onClick={() => { setCount(count - 1) }}>-</button>

          <span>{count}</span>

          <button onClick={() => { setCount(count + 1) }}>+</button>

        </div>

    )

}

Context

Context 提供了一种在组件之间共享此类值的方式，而不必显式地通过组件树的逐层传递 props。

主要提供一个提供者模式和消费者模式

代码样例:

import React, { createContext } from 'react';

import ReactDOM from 'react-dom';

import './index.css';

import \* as serviceWorker from './serviceWorker';

// 结构出提供者和消费者

const {

    Provider,

    Consumer:   CounterConsumer

} = createContext()

// 单独实现一个提供者类  用于数据共享   其实和vue 的bus 总线模式差不多

class CounterProvider extends React.Component {

    constructor() {

        super()

        this.state = {

            count: 100

        }

    }

    decrementCount = () => {

        this.setState({

            count: this.state.count - 1

        })

    }

    incrementCount = () => {

        this.setState({

            count: this.state.count + 1

        })

    }

    render() {

        return (

            <Provider value= {

                {

                count: this.state.count,

                decrementCount: this.decrementCount,

                incrementCount: this.incrementCount

                }

                }>

                { this.props.children }

            </Provider>

        )

    }

}

class CounterBtn extends React.Component {

    // 使用消费者的时候必须是个函数   函数参数就是传递的元素对象

    render() {

        return (

            <CounterConsumer>

            {

                (args) => {

                  const { decrementCount, incrementCount } = args

                  const handle = this.props.type === 'increment' ? incrementCount : decrementCount

                    return (

                      <button onClick = {handle}>{this.props.children}</button>

                    )

                }

            }

        </CounterConsumer>

        )

    }

}

class Counter extends React.Component {

    // consumer 里的值必须是函数

    render() {

        return (

          <CounterConsumer>

              {

                 (args) => {

                     const { count } = args

                    return (

                        <>

                         <span>{ count }</span>

                        </>

                    )

                 }

              }

          </CounterConsumer>

        )

    }

}

class App extends React.Component {

    render() {

        return(

            <div>

                <CounterBtn type="decrement">-</CounterBtn>

                <Counter />

                <CounterBtn type="increment">+</CounterBtn>

            </div>

        )

    }

}

// 用提供者标签包裹父标签

ReactDOM.render(<CounterProvider>

    <App />

</CounterProvider>,

 document.getElementById('root'));

// If you want your app to work offline and load faster, you can change

// unregister() to register() below. Note this comes with some pitfalls.

// Learn more about service workers: https://bit.ly/CRA-PWA

serviceWorker.unregister();

自定义实现react 状态管理功能

<html>

    <head>

    </head>

    <body>

      <div>

        <button onclick="store.dispatch({type: 'JIAN', n: 2})">-</button>

        <span id="countDispay"></span>

        <button onclick="store.dispatch({type: 'JIA', n: 3})">+</button>

      </div>

      <script>

        const countState = {

          count: 5

        }

        let countDispay = document.querySelector("#countDispay")

        const reduce = (state,action) => {

               if(!state){

                 return countState

                }

                switch(action.type) {

                  case 'JIAN' :

                  return {

                    ...state,

                    count: state.count - action.n

                  }

                  case 'JIA' :

                  return {

                    ...state,

                    count: state.count + action.n

                  }

                  break

                  default:

                    break

                }

        }

        const createStore = (reduce) => {

          let state = null

          // 获取状态值

          const getState = () => state

          let listeners = []

          // 订阅 将要执行的函数订阅到里面去

          const subcrime = (listener) => listeners.push(listener)

          // 触发改变状态

          const dispatch = (action) => {

            state = reduce(state,action)

            listeners.forEach((lister, index) => lister())

          }

          dispatch({})

          return {

            getState,

            dispatch,

            subcrime

          }

        }

        const store = createStore(reduce)

        const renderCount = () => {

          countDispay.innerHTML = store.getState().count

        }

        store.subcrime(renderCount)

        renderCount()

      </script>

    </body>

</html>

Redux 使用react-redux 来实现store 的跨组件绑定

入口文件:

import store from './store.js'

import { Provider } from 'react-redux'

ReactDOM.render(

<Provider store = { store }>

<App/>

</Provider>,

Reduce 的cart.js

import actionType from '../action/actionType'

const initailState = [{

    id: 1,

    title: 'apple',

    price: 888,

    amount: 20

},

{

    id: 2,

    title: 'phone',

    price: 5555,

    amount: 20

}]

// 默认给一个state 值

const cartReduce = (state = initailState, action) => {

         // 根据类型改变state 状态的值

         switch(action.type){

             case actionType.AMOUNT\_CART\_INCREMENT:

             return state.map((item) => {

                 if(item.id === action.payload.id){

                     item.amount += 1

                 }

                 return item

             })

             case actionType.AMOUNT\_CART\_DECREMENT:

             return state.map((item) => {

                 if(item.id === action.payload.id){

                     item.amount -= 1

                 }

                 return item

             })

             default:

                 return state

         }

}

export default cartReduce

合并reduce 文件的index.js:

// 使用combineReducers 来合并reduce

import {combineReducers} from 'redux'

import cart from './cart'

export default combineReducers({

    cart

})

Store.js

import { createStore } from 'redux'

import rootReduces from './redux'

const store = createStore(rootReduces)

export default store

Action.js 文件:

import actionType from './actionType'

export const increment = (id) => {

     return {

         type: actionType.AMOUNT\_CART\_INCREMENT,

         payload: {

             id

         }

     }

}

export const decrement = (id) => {

    return {

        type: actionType.AMOUNT\_CART\_DECREMENT,

        payload: {

            id

        }

    }

}

actionType.js 文件:

const actionType = {

    AMOUNT\_CART\_INCREMENT: 'AMOUNT\_CART\_INCREMENT',

    AMOUNT\_CART\_DECREMENT: 'AMOUNT\_CART\_DECREMENT'

}

export default actionType

导出action 的index.js 问价

import { increment, decrement } from './cart'

export {

    increment,

    decrement

}

cartList.js

import React, {Component} from 'react';

import { increment, decrement } from '../../action'

import { connect } from 'react-redux'

class CartList extends Component {

   /\* constructor() {

        super()

        this.state = {

            CartList: []

        }

    }

    // 获取改变之后的状态值并更新数据

    getState = () => {

        console.log(this.props.getState().cart)

        this.setState({

            CartList: this.props.getState().cart

        })

    }

    componentDidMount() {

       this.getState()

       // 订阅更新数据方法

       this.props.subscribe(this.getState)

    } \*/

    render() {

        console.log(this.props)

        return(

            <table>

                <thead>

                    <tr>

                    <th>标签</th>

                    <th>名称</th>

                    <th>单价</th>

                    <th>数量</th>

                    <th>操作</th>

                    </tr>

                </thead>

                <tbody>

                   {

                       this.props.cartList.map((item) => {

                           return (

                               <tr key={ item.id }>

                                   <td>{ item.id }</td>

                                   <td>{ item.title }</td>

                                   <td>{ item.price }</td>

                                   <td>

                                   <button onClick={this.props.decrement.bind(this, item.id)}>-</button>

                                   <span>{ item.amount }</span>

                                   <button onClick={ this.props.increment.bind(this, item.id)}>+</button>

                                   </td>

                                   <td></td>

                               </tr>

                           )

                       })

                   }

                </tbody>

            </table>

        )

    }

}

// 这个方法传递给connect 组件 利用这个方法将state 状态值挂在 props 属性上

const mapState = (state) => {

    return {

        cartList: state.cart

    }

}

// 这是一种拆解action 的方法， 直接将 action 挂在props

/\*const mapPropsFromAction = (dispatch) => {

    return {

        add: (id) => dispatch(increment(id)),

        reducer: (id) => dispatch(decrement(id))

    }

}

export default connect(mapState,mapPropsFromAction)(CartList) \*/

export default connect(mapState,{ increment, decrement })(CartList)

npm install redux --save 来使用redux

npm install react-redux --save 来实现redux 的非父子间通信功能

npm install redux-thunk --save 来实现 异步action redux 状态更新

使用applyMiddleware 来传递thunk

import { createStore, applyMiddleware } from 'redux'

import rootReduces from './redux'

import thunk from 'redux-thunk'

const store = createStore(rootReduces, applyMiddleware(thunk))

export default store

action的异步实现代码

export const decrementAsync = (id) => {

    return dispatch => {

        setTimeout(() => {

            dispatch(decrement(id))

        }, 2000)

    }

}

路由:

1. 安装： npm install react-router-dom --save

在index.js 中直接使用

ReactDOM.render(

<Router>

   <Route component={App} path="/" />

</Router>,

document.getElementById('root'));

其中跳转页面有link 组件 to 属性可以是一个对象

 <li><Link to="/home">首页</Link></li>

            <li><Link to={{

              pathname: '/article',

              state: {

                x: 1

              }

            }}>文章</Link></li>

            <li><Link to="/users">用户</Link></li>

这种类型叫做编程是路由

Route 有render 属性，可以动态渲染路由组件，传参等等

 <Switch>

          <Route component={Article} path="/article" exact/>

          <Route component={Home} path="/home" />

          <Route  path="/users" render={(routeProps)=> {

            return  this.state.isLogin ?

              <Users {...routeProps}/> :

              <div>请登录</div>

          }}/>

          <Route component={ArticleDetail} path="/article/:1" />

          <Redirect to="/home" from="/" />

          </Switch>

import React from 'react';

import { Route, NavLink as Link, Redirect, Switch } from 'react-router-dom'

import {

  Article,

  Home,

  Users,

  ArticleDetail

} from './views'

class App extends React.Component {

  state = {

    isLogin: true

  }

    render() {

      return (

        <div>

          <ul>

            <li><Link to="/home">首页</Link></li>

            <li><Link to={{

              pathname: '/article',

              state: {

                x: 1

              }

            }}>文章</Link></li>

            <li><Link to="/users">用户</Link></li>

          </ul>

          <Switch>

          <Route component={Article} path="/article" exact/>

          <Route component={Home} path="/home" />

          <Route  path="/users" render={(routeProps)=> {

            return  this.state.isLogin ?

              <Users {...routeProps}/> :

              <div>请登录</div>

          }}/>

          <Route component={ArticleDetail} path="/article/:1" />

          <Redirect to="/home" from="/" />

          </Switch>

        </div>

      )

    }

}

export default App

动态路由:

使用withRouter 来包裹，保证传参的时候将路由上的属性传到子组件上去,

this.props.history.push

这个方法直接跳转路由

import React from 'react';

import { withRouter } from 'react-router-dom'

class BackHome extends React.Component {

   goHome = () =>{

       this.props.history.push({

           pathname: '/home',

           state: {

               x: 1

           }

       })

   }

    render() {

      console.log(this.props)

      return (

        <div>

         <button onClick={this.goHome}>返回首页</button>

        </div>

      )

    }

}

export default withRouter(BackHome)