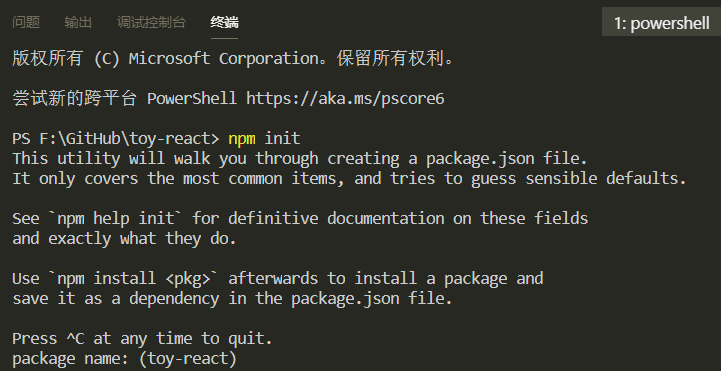
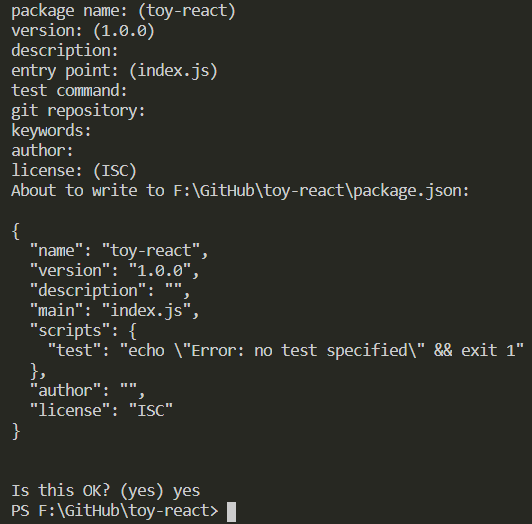
# Winter手把手带你实现ToyReact框架

## create

* 1. 新建一个名为toy-react文件夹，拖入到VS Code编辑器；
  2. 点击运行小图标；
  3. 点击，选择chrome选项；
  4. 打开VS Code控制台，选择终端，输入命令 npm init



* 1. 一路向下都是点击enter（注意：Is this OK? (yes) 此时输入yes）



* 1. 这时候步骤5会给你创建好一个package.json文件，可以看到有一个”main”:”index.js”，这是一个main的主的入口文件，主要是给node用的，我们并不是一个node项目所以在这里是无用的，但是仍然可以写一下main.js。

## Webpack Config

1. 在VS Code控制台使用npm install 来安装webpack，npm install webpack webpack-cli --save-dev

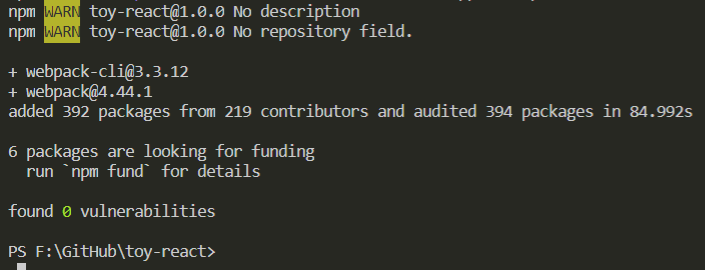
Webpack-cli：帮助我们去有命令行的一些基本的命令。

webpack和webpack-cli都属于在我们的开发过程中所使用到的工具而不是我们的代码本身，所以我们在npm上给一个--save-dev的指令，最终会加到package.json的devDependencies里面，不过这些在后面也是可以加的。

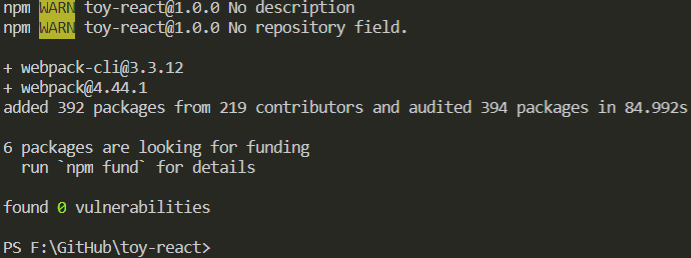
\*此处都是最基本最基本的安装方法，也可以选择别的，比如：yarn



1. webpack：
   1. 为什么要用webpack呢？
      1. Webpack其实是一个JavaScript的打包工具，它的输入和产出在正常的情况下都是JavaScript文件，它最大的作用就是帮助我们把JavaScript文件里面的import和require多文件打包成一个单个的js文件，所以webpack是由一个文件作为入口，这个文件可能会import一些东西，也可能会require一些东西，最终把它变成一个单个的大的文件，符合在web上的性能、发布各方面的一些的需求。Webpack同时还承载了很多的工具，比如Babel
      2. Babel是一个把新版本的js文件翻译成老版本的js文件的一种工具，Babel在webpack里面是以loader的形式去使用的，webpack允许使用loader去定制各种各样的文件，比如：原则上webpack只能打包普通的js文件，但是我们如果想把css的文件以某种形式打包成js文件的话，那么我们就可以写一个css-loader，如果想把html文件当作一个js文件打包进来，同样也可以写一个html-loader，而这个loader也可以是独立的包，只需要在webpack的配置里面配一下就可以了



1. Webpack已经安装成功了



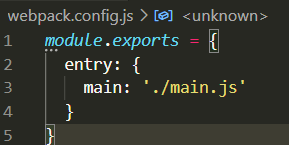


如果我们想要直接去使用webpack这个指令，还需要用npm -g去把他安装到全局，但是npm已经不太推荐这种npm install -g的一个行为了，阔以使用npx webpack来执行我们的webpack打包



因为没有去指定好webpack的config文件（webpack.config.js），虽然执行成功了，但是会抛出一个错误，Entry module没有找到，下一步就来把webpack的配置给它配好。

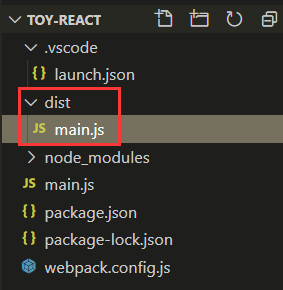
1. webpack.config.js文件
   1. 先创建一个名叫webpack.config.js文件
   2. 文件最主要的几个部分：
      1. 首先是Entry的部分
         1. 首先是需要给它指定一个入口的文件，是用node.js的标准的module.exports这样的一个写法，因为没有办法在对webpack本身去做Babel的转换，所以就干脆使用node的标准module.exports的写法
         2. 阔以写一个main.js，这个才是真正打包的一个入口文件，有了entry之后webpack就可以成功执行了



1. main.js
   1. 先新建一个空的main.js文件，然后让webpacl去执行
   2. 输入命令 npx webpack



* 1. 完成打包，打包成功之后，会出现dist目录

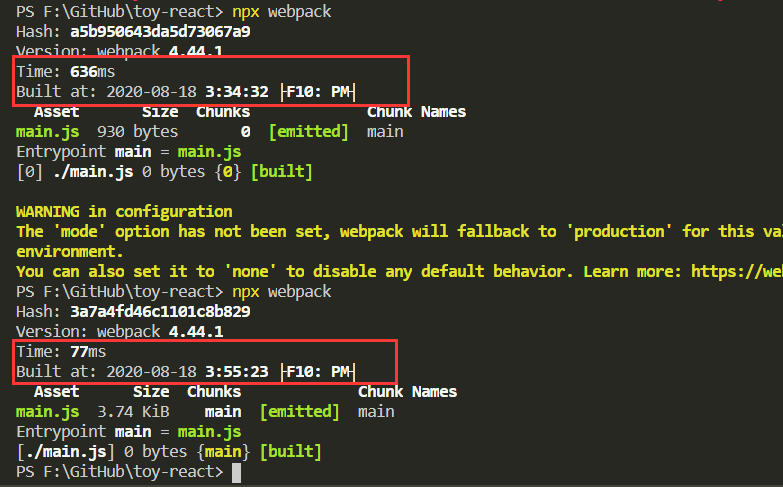


* 1. 由于dist里面的main.js是一段没人能看懂的压缩代码，所以我们为了方便调试可以在module.exports里面去加两个配置，让它build出来的文件成为一个可以被人类读懂的代码
     1. Develop mode设为development
     2. Optimization minimize设为false



有了这两个选项之后，它build出来的文件可读性就会大大的提升

* 1. 重新执行npx webpack



1. 这个时候dist里面生成的会发现不再对文件进行压缩了，并且在注释里面可以看到main.js的位置

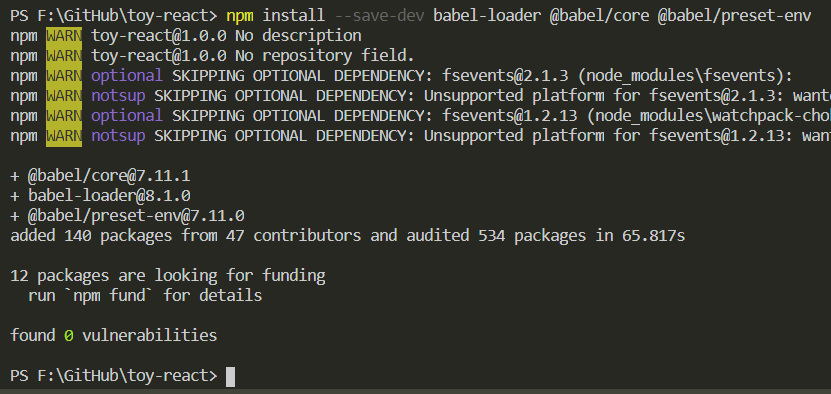


1. 而此时main.js里面现在是一个完全空的文件，用了eval去执行，执行之后后面带了一个sourceURL的一个部分，因为带了sourceURL我们在浏览器里如果打开这个main.js的时候，那么eval的部分就会被映射成一个单独的文件，这样webpack就能给我们提供一个非常易于人类阅读的一个调试的版本
2. Loader

webpack最常见也是最广泛应用的一个loader --> babel-loader。Babel可以把最新的js文件版本转换成老的版本，既可以写一些新的语言特性，又可以跑在较老的浏览器上，兼容性也会变得比较好，Babel也可以独立使用，去编译js文件

* 1. 先安装babel-loader的一个包
  2. 同时安装@babel/core包

Babel本身不带任何配置的一个核心，为了安装一个比较常用的配置选项

* 1. 同时还需要安装一个preset-env的一个包，都是dev里面的内容 
  2. Babel安装完毕，可以package.json文件里面查看一系列的包

## 写webpack的module配置

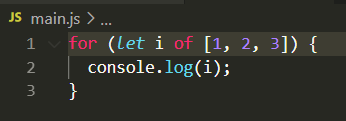
* 1. 写上一个test和一个use属性



* 1. 给babel-loader添加配置

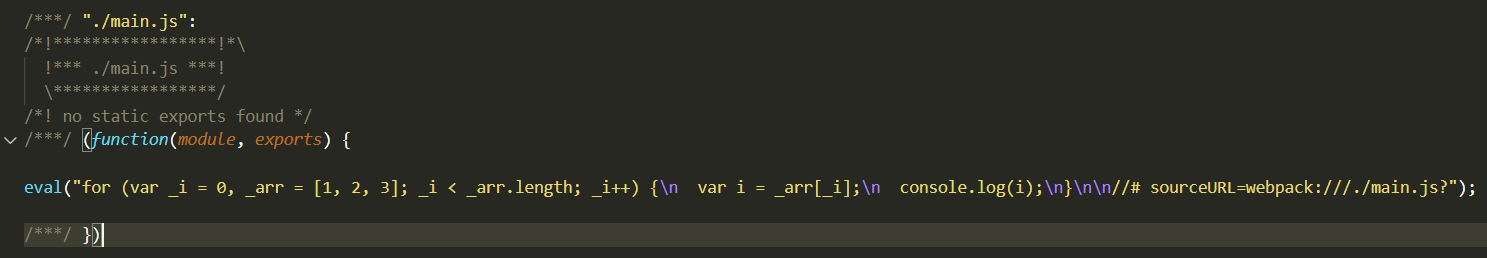


* 1. 配置完毕了，在main.js里面

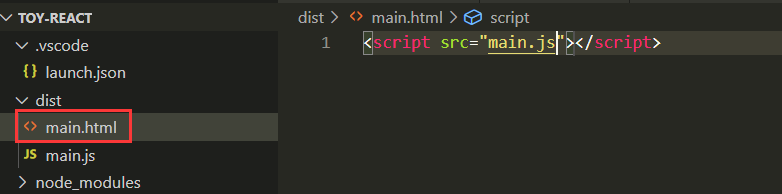


执行npx webpack

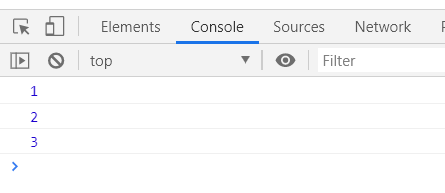
dist目录下的main.js文件发生了变化

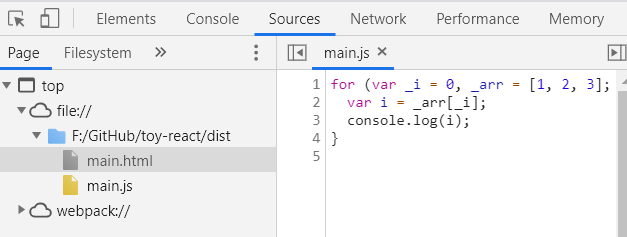


* 1. 在dist目录下面添加一个html文件用来跑main.js文件

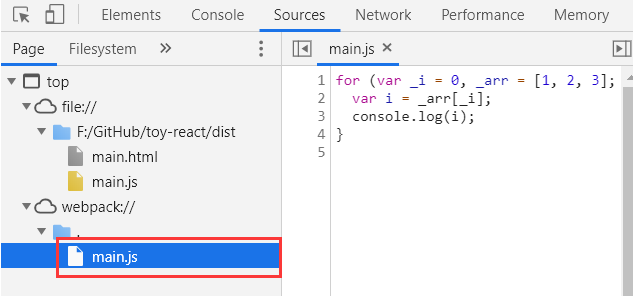


* 1. 去到资源管理器中直接打开main.html



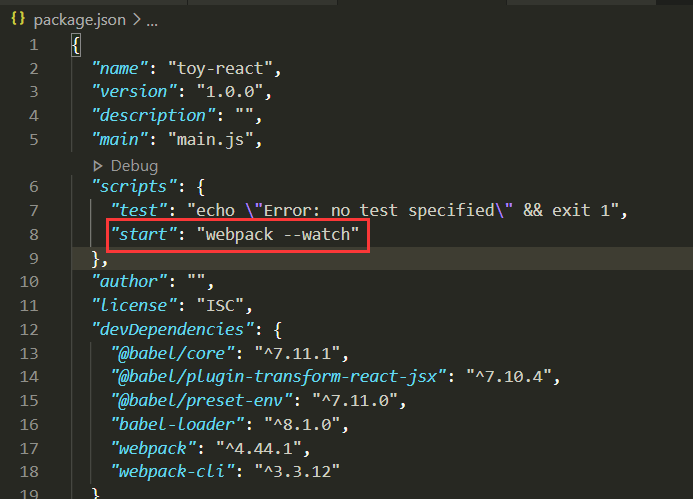


* 1. 会看到有一个虚拟的main.js文件，正是Babel为我们造出来的虚拟文件

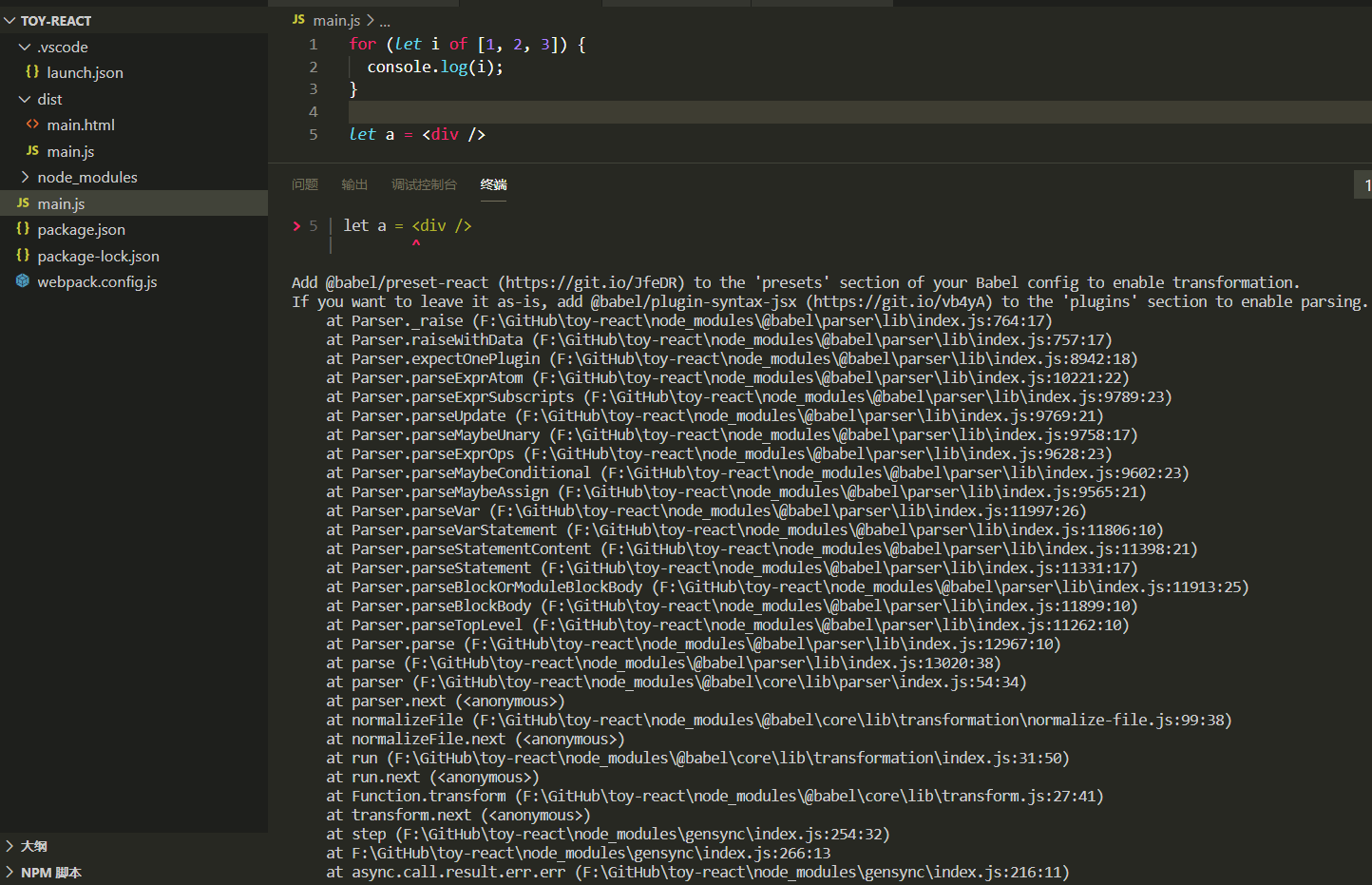


* 1. 完成babel的插件的配置

我们可以额外的在package.json里面的scripts下面配置一个start:webpack --watch，这个的好处是不用每次执行npx webpack命令刷新浏览器，而是直接保存文件刷新浏览器就好



* 1. 在main.js里面写一个JSX

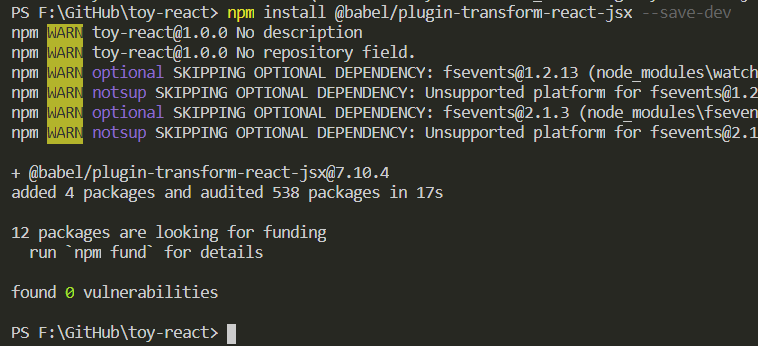


抛出错误，因为我们在Babel的正常的preset里面它是不包含JSX的能力的，此时需要npm安装

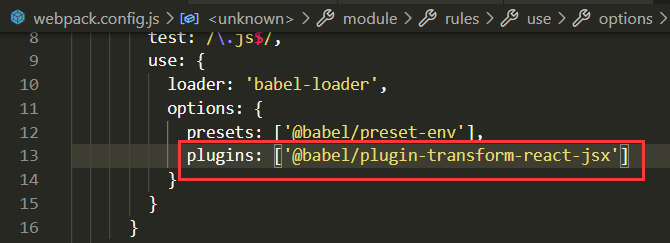
* 1. Babel里面的一个专门用来处理JSX的一个plugin，跟preset的区别：

一个preset里面可能会包含若干个plugin，名字叫做plugin-transform-react-jsx，

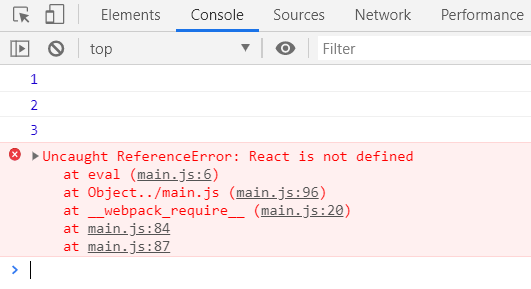
如果忘记--save-dev参数，就会安装到普通的dependencies里面，需要手动的把package.json改对

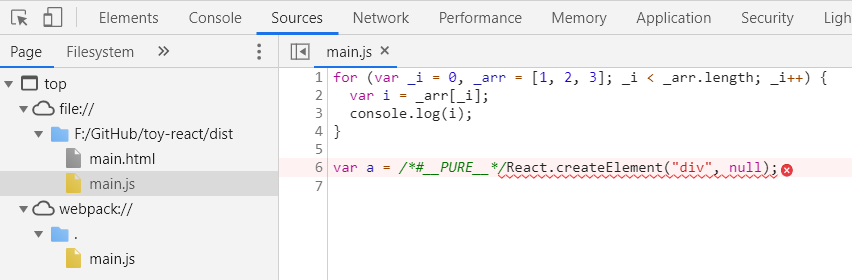


* 1. 由于是一个babel的plugin，配进去的时候要改babel的options，babel的options里面可以放presets，也可以放plugin，babel的plugin通过它的config里的plugins属性，也是一个数组



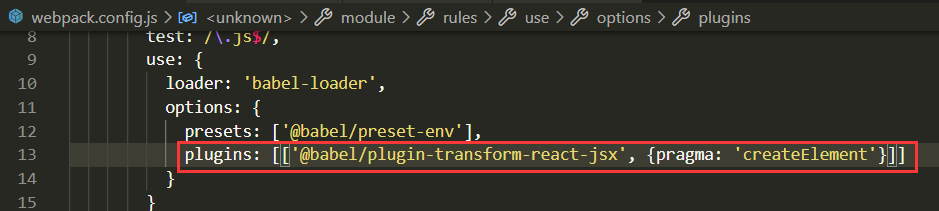
* 1. 先前编译抛出的错误现在可以顺利通过了



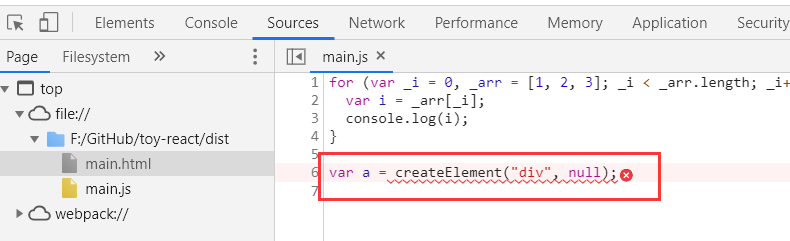


直接翻译出来了一个带react的一个函数名字，怎么变成toy-react呢？

* 1. 修改plugin的设置

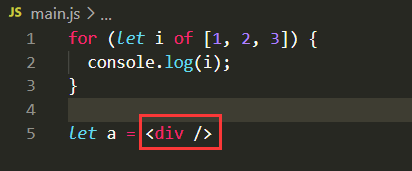


执行npx webpack



Pragma里面什么最后就会transform出什么来（比喻将createElement改成chengyafang，想知道结果吗）

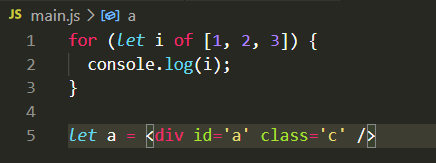
* 1. 不难发现JSX环境里面其实一个非常非常重要的就是div会被翻译成一个函数调用



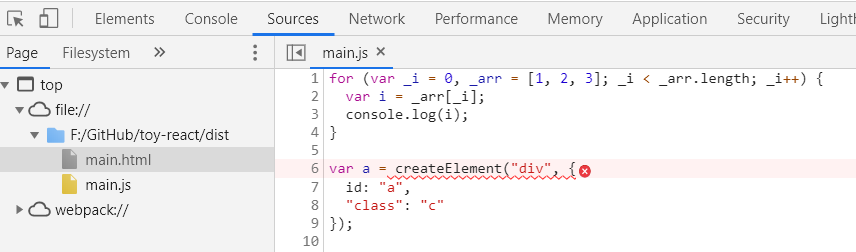
* 1. 环境部分搭建完成

## 如何去实现JSX翻译出来的函数

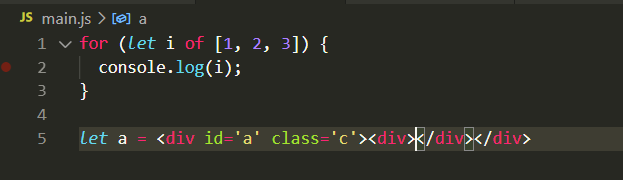
* 1. 真正的JSX到底能产生什么样的东西
     1. 给div添加一个属性



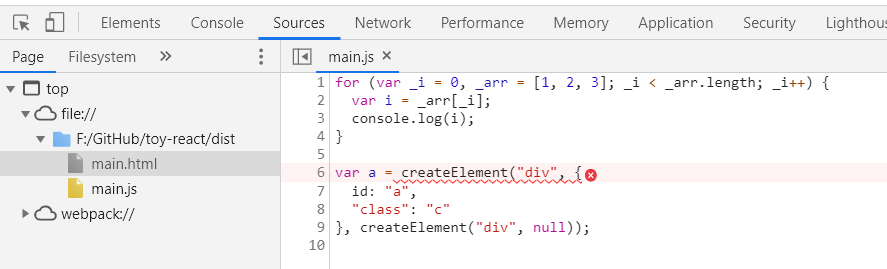
运行结果为：



* 1. 添加子节点



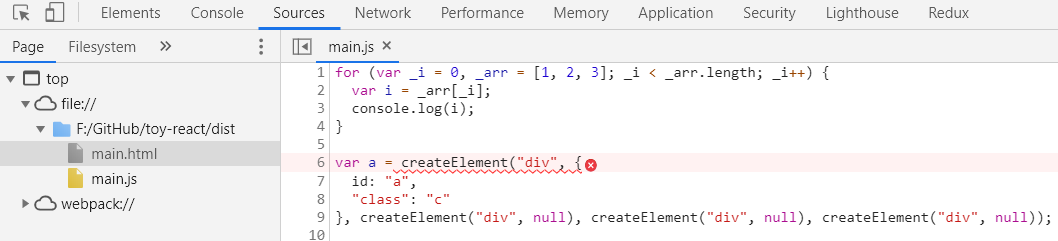
运行结果为：



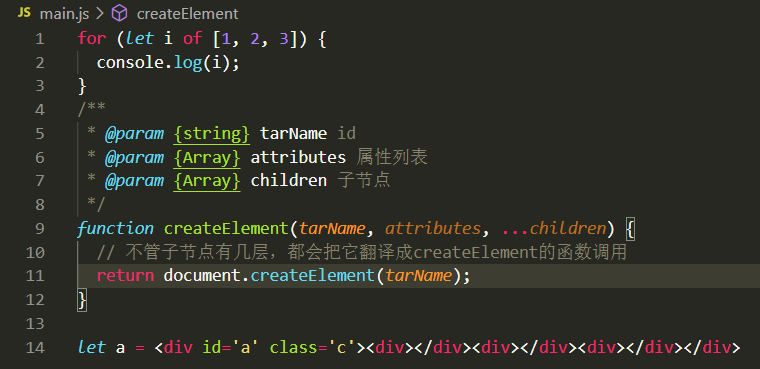
* 1. 多个子节点



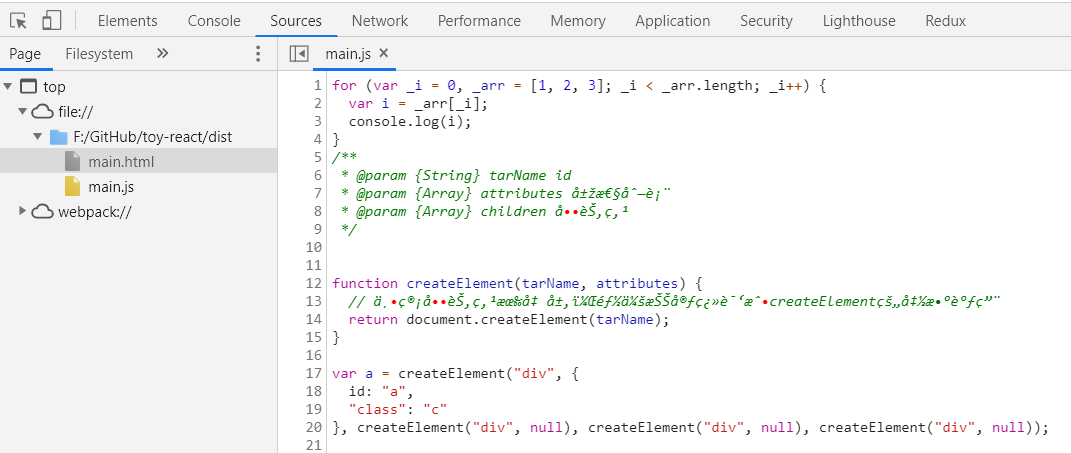
我们可以发现，有三个参数，但是不管有几个children都是依次的把它作为参数传递到它的后面的参数里面



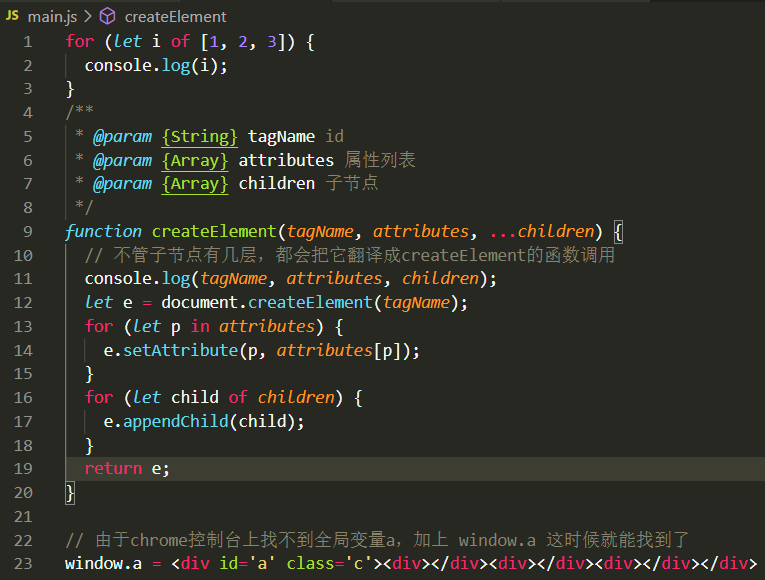
* 1. 要想实现一个createElement应该怎么做？



运行结果：可以看到已经不报错了



* 1. 将main.js里面的a变成全局变量



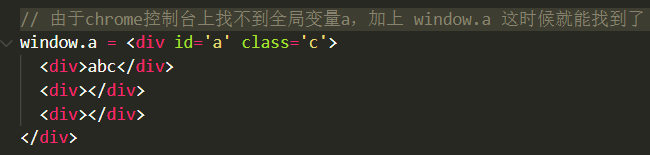
在chrome控制台上输入window.a



此时就变成了一个有结构的window的一个DOM element，这个时候JSX已经具有初步的可用性了，不过还是没有考虑处理文本节点

* 1. main.js：

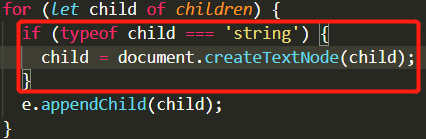
如果我们在div里面加一个abc节点



此时浏览器控制台会报错

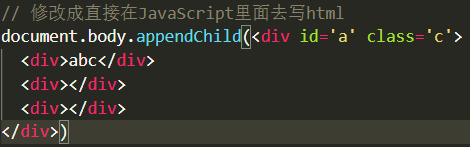


由于传入的节点是一个字符串，所以给他一个判断逻辑

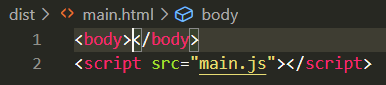




* 1. main.js

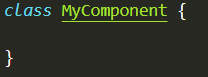


在dist目录下的main.html里面需要加上body标签

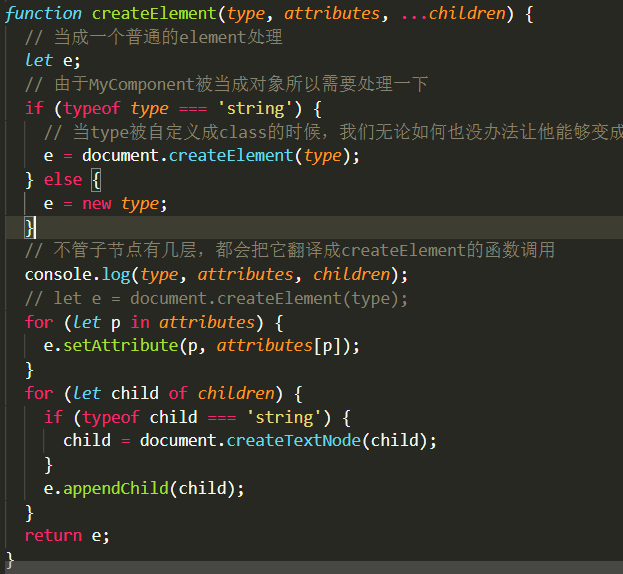


这样就直接渲染在浏览器页面上

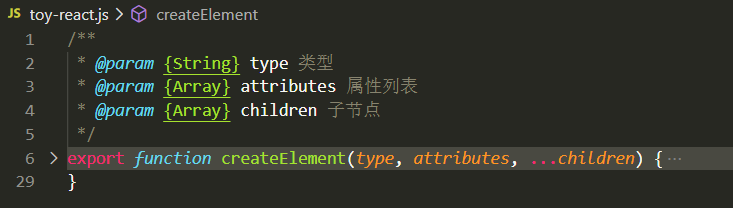
* 1. 瞅瞅jsx里面的自定义组件机制是什么样子？
     1. 当我们想document.body.appendChild里面的div元素变成组件的时候就需要新建一个名为MyComponent组件，一般自定义组件默认为class，也可以是对象，根据最新的host API是函数；由于我们不能修改原生方法，所以接下来我们需要自己封装方法。



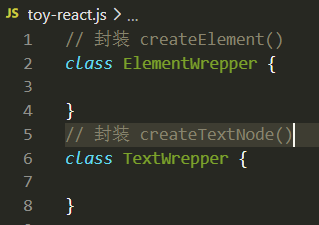
* + 1. 这时候函数createElement函数里面的参数因为需要封装将tagName改为type



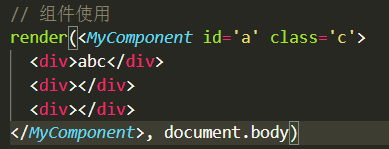
* 1. 新建toy-react.js，我们将封装函数createElement函数挪到此文件中



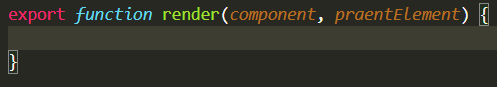
* + 1. 需要封装的createElement、createTextNode方法
    2. wrapper：给两个封装方法改写



* + 1. 在main.js中我们用一个render函数来包裹组件



* + 1. 在toy-react.js中添加render函数



* + 1. 给步骤（1）添加内容、在main.js里面MyComponent添加内容
    2. MyComponent组件如果没有改写方法就默认继承



* + 1. 实现ElementWrepper
    2. 虽然简易的toy-react已经完成，但是还缺少自定义组件react所具有的state、生命周期、一系列能力

## 补全React

1. 数据来源
   1. 自定义组件能够支持state
   2. setState不但改变了state这个值本身同时它会启动一个重新render的这样的一个动作，然后当你多次setState之后，它会在整个生命周期结束的时候，发起一次重新的render，而这次render之后组件就会按照正确的模板去写了。
   3. 改写MyComponent自定义组件
2. 数据更新
   1. Render环节
   2. 取root的过程
   3. 完成了基于range的DOM绘制

## 重新绘制（虚拟DOM）

1. 重新绘制
   1. 存range才能重新绘制
   2. 用rerender重新绘制算法
   3. 调用rerender方法
2. 渲染能力
   1. setState()

## 尝试把React的Tutorials里面的内容跑起来



根据react教程的一个三子棋游戏改写

1. Range引入的bug（鼠标先点击左边方格，再点击右边方格的时候方格会消息不见），也就是从左往右没问题，从右往左会丢东西。
2. Rerender改写

## React怎么解决引入VDOM并且实现VDOM

由于没有引入虚拟dom，所以每次都是全量的去更新它的整个dom树，在react的这种UI等于数据加上一个模板的思路下，如果不引引入Virtual Dom那么就只能全量更新DOM树，过程中对性能的损耗是非常大的，哪怕只做一个非常细小的操作，最后一定会导致有一个非常大规模的DOM更新，这样是不能接受的，所以需要引入VDOM

## 将虚拟DOM变成实体DOM

如何从一个已有的虚拟dom树和已有的实体dom树之间生成一个patch逻辑

## 如何实现VDOM的比对

把一棵树新的vdom跟上次渲染得到的旧vdom去做一个对比，然后决定哪一个树的子树去做重新渲染