# 前端工程化~上篇

《WEB实训》

# 目录

# 为什么要工程化？~问题

- 每引入一个模块就要去找CDN链接或者下载包，未免也太低效了？
- 正在使用的ES6，浏览器并不支持，要发布出去，必须先做一层转换，这个转换怎么做？
- 应用越来越复杂，组件越来越多，肯定是会有很多个JS文件，CSS文件，怎么让这些文件相互协作？
- 开发时要方便调试，方便比性能重要；发布后则相反，性能比方便重要，安全比调试重要，如何同时兼顾这两方面？
- 代码需要遵守代码规范，怎么约束开发者遵守规范？
- 。。。

# 为什么要工程化？~答案

- “工欲善其事，必先利其器”，要开发的功能可以理解为“事”，工程化可以理解为“器”，工程化没做好，事情做起来也不会舒服。
- 前端工程化具体要做的，就是将前端的开发流程、技术、工具、经验等规范化、标准化。它的目的是让前端开发能够自成体系，最大程度地提高前端工程师的开发效率，降低技术选型、前后端联调等带来的协调沟通成本。

# 前端工程化要掌握什么?

- 包管理：比如npm或yarn, 解决引入react, axios, font-awesome等各种各样的包不方便的问题。
- 编译器：比如Babel,主要解决使用了ES6或者更新的语法，在旧浏览器上可以正常运行的问题。
- 打包器：比如webpack,主要解决JS模块化以后，分散到各个目录的JS文件，CSS文件，图片等等如何将打包成一个完整的APP的问题。
- 分环境配置：开发环境、生产环境、测试环境，它们的包、编译器、打包器用的配置可能都不一样，需要能够根据环境需要配置。
- 压缩、混淆、文件名哈希：生产环境要求的配置
- 工程目录规划：文件放哪里，是个令初学者十分头疼的问题，应参考别人的脚手架
- 语法检查：eslint自动检查语法
- 。。。

# 包管理器~作用

- 核心功能第一是统一和简化包的安装，比如前面说的react, axios,font-awesome都不用到网上找，直接输入npm install react axios font-awesome就安装好了。

- 核心功能第二是解决包的依赖，一个大型的应用，可能有几百个包，某个版本的包依赖另一个版本的包，版本用错就不能正常工作，如果靠手动检查比对，那绝对是要崩溃的。

- 包管理还有执行脚本的功能，它不是必须的，但是提供了许多便利。一般用来启动开发环境，生成目标文件，执行测试、做lint检查等等。

# 包管理器~工具

- 包管理器使用npm或者yarn，两者都要掌握。其中，npm是官方的包管理器，必须掌握；但是yarn比npm快很多，我们常常用yarn代替npm。为便于讨论，后续只讲npm，yarn请自行查文档了解。
- 还有其他一些包管理器像bower，已经不用去了解了

# 包管理器-npm命令用法

- npm init 初始化包基本配置
- npm install 安装依赖包
- npm uninstall 卸载依赖包
- npm update 更新依赖包

# 包管理器-npm init

## 初始化新项目

```
$npm init
```

初始化会生成一个package.json文件，这些信息是你发布自己的包时必须的。我们现在的目标是使用别人的包，生成 的这些信息都不是必须的，简单了解下即可。

## 命令执行后根据提示交互

```
Press ^C at any time to quit.
package name: (demo1)
version: (1.0.0)
description: Demo Project
entry point: (index.js)
test command:
git repository: https://github.com/pheye/train
keywords: React,Frontend,Train
author: LIUWENCAN
license: (ISC) MIT
About to write to /Users/liuwencan/src/react-train/demo1/package.json:

{
  "name": "demo1",
  "version": "1.0.0",
  "description": "Demo Project",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "repository": {
    "type": "git",
    "url": "git+https://github.com/pheye/train.git"
  },
  "keywords": [
    "React",
    "Frontend",
    "Train"
  ],
  "author": "LIUWENCAN",
  "license": "MIT",
  "bugs": {
    "url": "https://github.com/pheye/train/issues"
  },
  "homepage": "https://github.com/pheye/train#readme"
}
```

# 包管理器-npm init

## package.json的字段说明

- **name** - 包名。
- **version** - 包的版本号。
- **description** - 包的描述。
- **homepage** - 包的官网 url 。
- **author** - 包的作者姓名。
- **contributors** - 包的其他贡献者姓名。
- **dependencies** - 依赖包列表。如果依赖包没有安装，npm 会自动将依赖包安装在 node_module 目录下。
- **repository** - 包代码存放的地方的类型，可以是 git 或 svn，git 可在 Github 上。
- **main** - main 字段指定了程序的主入口文件，require('moduleName') 就会加载这个文件。这个字段的默认值是模块根目录下面的 index.js。
- **keywords** - 关键字

# 包管理器-npm install

常见用法的语法：

```
npm install # 根据package.json安装
npm install <package> # 安装指定包
npm install <package>@<version> # 安装指定版本的包
```

所有安装的包都在node_modules/目录下，同时会在package.json的dependencies字段下面会出现该包的信息。

示例1：

```
$npm install react
```

1.安装完以后，将出现node_modules/react目录，在里面可以找到react.development.js和react.production.min.js，试试将HTML中引用react的CDN链接改为本地路径。

2.打开package.json，可以在dependencies属性，找到 "react": "^16.11.0"这样的信息。

# 包管理器~包版本问题

- 当你打开package.json看react的版本时，上面写着^16.10.0,^是什么意思?
- npm install <package>@<version>，通过<version>指定版本，版本格式应该是怎样的？是写1.0.0,还是12345，或者a.b.c，能不能随便写?

这些问题都由语义化版本(SemVer)规范解决。该规范不仅在前端使用，在后端如PHP的包管理器也会使用。语义化版本的完整规则见下面链接。这里只展示示例出现的版本规则。

语义化版本的范围： https://www.jianshu.com/p/d306ed03de62

# 包管理器~语义化版本规则

## 语义化版本格式

我们首先简单了解一下语义化版本版本号，标准的版本格式为：X.Y.Z，其中：

- X：主版本号，当我们做了不兼容或者颠覆性的更新，修改此版本号。
- Y：此版本号，当我们做了向下兼容的功能性修改，修改此版本号。
- Z：修订号，当我们做了向下兼容的问题修正，修改此版本号。
- 其中X、Y和Z必须为非负整数，禁止数字前补零，每个数值都是递增的。

## 语义化版本范围

版本范围是一组满足指定范围的比较器，一个比较器是由操作符和版本号组成，下面是最原始的操作符：

- < 小于；
- <= 小于等于；
- > 大于；
- >= 大于等于；
- = 等于；如果没有指定操作符，则默认为等于。

一个范围可由一个或者多个比较器组成，如果有多个，则由双竖线（||）连接。对于包含多个比较器，只要满足其一即可。比如：

- 范围 >=1.2.7 <1.3.0 ，版本号 1.2.7, 1.2.8, 1.2.99 满足条件，而 1.2.6, 1.3.0 ,1.1.0 确不满足。
- 范围 1.2.7 || >=1.2.9 <2.0.0 ，版本号 1.2.7, 1.2.9, 1.4.6 满足，而 1.2.8 或者 2.0.0 不满足。

## 补注号（^）范围 ^1.2.3 ^0.2.5 ^0.0.4

允许在不修改[major, minor, patch]中最左非零数字的更改。换句话说，允许在 1.0.0 及以上版本对次版本号和修订版本号的更新，允许在 0.1.0以及以上版本对修订版本号更新，版本为 0.0.X 不允许更新。

^1.2.3 := >=1.2.3 <2.0.0

^0.2.3 := >=0.2.3 <0.3.0

^0.0.3 := >=0.0.3 <0.0.4

^1.2.3-beta.2 := >=1.2.3-beta.2 <2.0.0

^0.0.3-beta := >=0.0.3-beta <0.0.4

^1.2.x := >=1.2.0 <2.0.0

^0.0.x := >=0.0.0 <0.1.0

^0.0 := >=0.0.0 <0.1.0

^1.x := >=1.0.0 <2.0.0

^0.x := >=0.0.0 <1.0.0

# 包管理器~包版本问题2

- package.json里面dependencies说明的既然是依赖包的版本范围，那实际安装的包具体是哪个版本?

该问题由package-lock.json解决，当包安装以后，npm会在package-lock.json中写明实际安装的版本。

# 包管理器~包的来源问题

- npm install的包是从哪里获取的?

答案npmjs.com，比如react的包，就在：

https://www.npmjs.com/package/react

npmjs.com是国外的网站，下载十分缓慢，于是国内就出现了许多镜像站，每几十分钟去同步官方源。当安装很慢时，要修改npm去从淘宝NPM镜像源获取包。具体方法见官方文档，使用cnpm代替npm是最快的方法：　http://npm.taobao.org/

# 包管理器-npm uninstall

语法：

```
npm uninstall <package> # 删除指定包
```

说明：将从package.json, package-lock.json删除该包的信息。同时node_modules/目录下的包也会被删除。

示例：

```
$npm uninstall react
```

# 包管理器-npm update

语法：

```
npm update # 更新所有包
npm update <package> # 更新指定包
```

说明：不带参数的npm udpate会检查package.json的所有包，能更新的将做更新，并修改package.json的版本规则，以及更新package-lock.json。而npm update <package>的唯一区别是只检查指定包而不是所有包。

示例：

```
$npm update react
```

# 包管理器~综合示例

## 对上周作业做一下改造

```
$npm install react
$npm install react-dom
$npm install axios
$npm install font-awesome
$npm install babel-standalone
```

1.先安装包

2.然后修改HTML文件，引用本地包

```
<!DOCTYPE html>
<html lang="en">

<head>
    <meta charset="UTF-8">
    <title>Github热门项目</title>
    <style>
        /* 这块由normalize.css处理   */
        body {
            padding: 0;
            margin: 0;
        }
        a {
            text-decoration: none;
        }
    </style>
    <link href="node_modules/font-awesome/css/font-awesome.min.css" rel="stylesheet">
</head>

<body>
    <div id="app"></div>
    <script src="node_modules/react/umd/react.production.min.js"></script>
    <script src="node_modules/react-dom/umd/react-dom.production.min.js"></script>

    <script src="node_modules/axios/dist/axios.min.js"></script>
    <script src="node_modules/babel-standalone/babel.min.js"></script>

    <script type="text/babel" src="scripts/app.js">
    </script>
</body>

</html>
```

# 编译器~介绍

- 编译器主要指Babel工具链，它的作用是将ES6及以上的语法，转换成向后兼容的JS语法，以便程序能运行在当前和旧版本的浏览器或其他环境中。比如下面这个例子，箭头函数许多浏览器不支持，经过一层转码后浏览器就都支持了：

```
// 转码前
input.map(item => item + 1);

// 转码后
input.map(function (item) {
  return item + 1;
});
```

# 编译器- babel-standalone的问题

- babel支持的使用方式非常广泛，前面引入了babel-standalone是最简单的一种，但是这种用法只能在开发环境中用，在生产环境中使用会引起严重的性能问题。实际的生产环境需要提前用babel对源码编译，直接使用编译后的源码，一般使用命令行或者配合打包器使用。此节先看命令行的使用，在打包器一节将谈到如何配合。

# 编译器~命令行方式

## 第一步：安装babel的核心包和命令工具

```
$npm install @babel/core # babel核心
$npm install @babel/cli # babel命令行工具
```

## 第二步：安装babel的预置配置和语法插件

```
$npm install @babel/preset-env # ES6语法
$npm install @babel/preset-react # React语法
$npm install @babel/plugin-proposal-class-properties # 支持类属性，后面解释
```

## 第三步：编辑.babelrc

```
{
    "presets": [
        [
            "@babel/preset-env",
        {
            "useBuiltIns": "entry",
            "corejs": 3
        }
        ],
        "@babel/preset-react"
    ],
    "plugins": [
        "@babel/plugin-proposal-class-properties"
    ]
}
```

# 编译器~命令行方式

第四步：安装babel-polyfill

```
$npm install @babel/polyfill
```

第五步：修改html,删除babel-standalone,

引入polyfill，修改JS路径（绿色字体代码）

```
<body>
    <div id="app"></div>
    <script src="node_modules/react/umd/react.production.min.js"></script>
    <script src="node_modules/react-dom/umd/react-dom.production.min.js"></script>

    <script src="node_modules/axios/dist/axios.min.js"></script>
    <script src="node_modules/@babel/polyfill/dist/polyfill.min.js"></script>

    <script src="dist/app.js">
    </script>
</body>
```

第六步：执行babel编译,确认应用是否能正常运行

```
$npx babel scripts/ --out-dir dist  # 将生成dist目录，里面有编译后的js文件
```

*每次修改JS代码或者babel配置，就需要重新执行第六步。

# 编译器~命令行方式说明~0

- 从命令行的步骤就可看出，整个配置非常繁琐，一个想用ES6+语法写React的人，还没写出第一个React就得学习这么复杂的配置是非常不友好的。这便是React基础课程直接使用babel-standalone的原因，到现在React的应用做出来了，再来优化babel的使用是更恰当的选择。下面我们详细分析上面步骤的各个步骤。

# 编译器~命令行方式说明~1

## 第一步：安装babel的核心包和命令工具

```
$npm install @babel/core # babel核心
$npm install @babel/cli # babel命令行工具
```

说明：执行完这一步，其实就可以执行第六步了，但是一般都会报错。babel本身只是个壳，用于执行其他babel插件，完成对ES6+和JSX的实际解析。第二步便是安装实际解析的插件。为什么Babel要搞得这么麻烦呢？这是因为babel不仅要运行在浏览器端，还要用在node端，而eslint做代码检查或者jest做单元测试也需要使用，把这些插件的支持都塞一起babel会很大，同时它们的配置上也有一定的区别，同样需要靠配置去区分；另外，JS规范本身也在不断发展，我们说ES6的时候，其实是指ES6+，即ES6,ES7等等，像async/await也并不是一开始就是ES6里面的，而是规范不断发展新增的结果，这里面有许多实验特性后来又被废弃，如果babel都集成在一起，而不是以插件的形式提供，就无法解决这方面的问题。

# 编译器~命令行方式说明~2

## 第二步：安装babel的预置配置和语法插件

```
$npm install @babel/preset-env # ES6+语法
$npm install @babel/preset-react # React语法
$npm install @babel/plugin-proposal-class-properties # 支持类属性
```

说明：ES6和React的语法，实际是由预设和插件解析的，babel加载不同的预设和插件，就能支持做任意多种的插件。由此也不难推出，babel的插件可能会有几十上百种。这也是预设存在的目的。

正常情况下，我们引入@babel/preset-env和@babel/preset-react预设就认为支持ES6和JSX之类的语法了。其他的插件，则是通过错误判断。比如@babel/plugin-proposal-class-properties，一开始我们是没引入的，直接执行第六步，结果报了下图的错误。将这个错误百度一下，就是由于没加载该插件引起的，于是安装下该插件就行了。如果还有错误，就继续用同样的方式解决。

# 编译器~命令行方式说明~2

```
{ SyntaxError: /Users/liuwencan/src/react-train/demo2/scripts/app.js: Support for the experimental syntax 'classProperties' isn't currently enabled (173:12):

  171 |         }
  172 |     }
> 173 |     search = async () => {
      |            ^
  174 |         const {query} = this.props;
  175 |         const url = `https://api.github.com/search/repositories?q=${query}&sort=stars&order=desc&type=Repositories`;
  176 |         console.log('url', url);

Add @babel/plugin-proposal-class-properties (https://git.io/vb4SL) to the 'plugins' section of your Babel config to enable transformation.
    at Object.raise (/Users/liuwencan/src/react-train/demo2/node_modules/@babel/parser/lib/index.js:6930:17)
    at Object.expectPlugin (/Users/liuwencan/src/react-train/demo2/node_modules/@babel/parser/lib/index.js:8328:18)
    at Object.parseClassProperty (/Users/liuwencan/src/react-train/demo2/node_modules/@babel/parser/lib/index.js:11617:12)
    at Object.pushClassProperty (/Users/liuwencan/src/react-train/demo2/node_modules/@babel/parser/lib/index.js:11582:30)
    at Object.parseClassMemberWithIsStatic (/Users/liuwencan/src/react-train/demo2/node_modules/@babel/parser/lib/index.js:11516:14)
    at Object.parseClassMember (/Users/liuwencan/src/react-train/demo2/node_modules/@babel/parser/lib/index.js:11453:10)
    at withTopicForbiddingContext (/Users/liuwencan/src/react-train/demo2/node_modules/@babel/parser/lib/index.js:11408:14)
    at Object.withTopicForbiddingContext (/Users/liuwencan/src/react-train/demo2/node_modules/@babel/parser/lib/index.js:10486:14)
    at Object.parseClassBody (/Users/liuwencan/src/react-train/demo2/node_modules/@babel/parser/lib/index.js:11385:10)
    at Object.parseClass (/Users/liuwencan/src/react-train/demo2/node_modules/@babel/parser/lib/index.js:11359:22)
  pos: 4560,
  loc: Position { line: 173, column: 11 },
  missingPlugin: [ 'classProperties' ],
  code: 'BABEL_PARSE_ERROR' }
```

这个错误是因为search = async () => {...}这种写法，是在ES7引入的。

# 编译器~命令行方式说明~3

## 第三步：编辑.babelrc

```
{
    "presets": [
        [
            "@babel/preset-env",
        {
            "useBuiltIns": "entry",
            "corejs": 3
        }
        ],
        "@babel/preset-react"
    ],
    "plugins": [
        "@babel/plugin-proposal-class-properties"
    ]
}
```

说明：预设和插件安装以后，只是放在node_modules目录下，babel会不知道要去加载它们，.babelrc就是告诉babel要去加载哪些预设，哪些插件，以及修改它们支持的选项。从上面的代码可以看到，我们上一部安装的预设和插件都在这里用上了。同时，修改了@babel/preset-env的"useBuiltIns"为"entry"，这个选项是用来与下一步的polyfill配合使用的，在下一步详细解释。

# 编译器-命令行方式说明-4

## 第四步：安装babel-polyfill

```
$npm install @babel/polyfill
```

说明：babel默认只转换语法，不转换对象。转换语法是对代码做个变换即可，转换对象的话，由于原浏览器没有这些对象的源码，babel需要往里面插入一些新的对象，像Promise, Symbol。像async/await会被转换为使用Promise，如果不对babel的配置做一些调整，在运行的时候就会报错。

解决这个问题有两种：

1.一种是使用@babel/polyfill，在HTML引入JS文件，放在自己的代码之前就行了，使用方便，但是有风险。它的原理是自己实现了ES6+的所有对象和实例方法，污染了全局空间和内置对象，导致产生同名对象冲突;

2.另一种使用@babel/runtime，babel检测到使用了哪些ES6的对象，按需引入需要的对象，不会污染全局空间和内置对象，这种方法更智能，实际生产推荐使用，但是配置较复杂，需要与打包器webpack配合；

所以这里我们先使用第1种解决问题。

第1种方法就是2步，

第1步：修改了@babel/preset-env的"useBuiltIns"为"entry"，babel会将ES6对象转换成对core-js的引用;

第2步：在HTML中引入babel-polyfill的文件（该文件其实就是core-js加其他额外的对象），下一步就是做这事;

babel-polyfill与babel-runtime深入了解：https://blog.csdn.net/weixin_34151004/article/details/93175123

# 编译器~命令行方式说明~5

## 第五步：修改html,删除babel-standalone,引入polyfill，修改JS路径（绿色字体代码）

```
<body>
    <div id="app"></div>
    <script src="node_modules/react/umd/react.production.min.js"></script>
    <script src="node_modules/react-dom/umd/react-dom.production.min.js"></script>

    <script src="node_modules/axios/dist/axios.min.js"></script>
    <script src="node_modules/@babel/polyfill/dist/polyfill.min.js"></script>

    <script src="dist/app.js">
    </script>
</body>
```

说明：

1.引入polyfill，上一步已经说过，不再赘述。

2.使用<script src="dist/app.js"/>直接引用编译后的文件。

# 编译器~命令行方式说明~6

## 第六步：执行babel编译,确认应用是否能正常运行

```
$npx babel scripts/ --out-dir dist  # 将生成dist目录，里面有编译后的js文件
```

说明：这一步就是执行babel命令，生成前一步需要的dist/app.js。每改一次scripts/的文件或者babel配置，都需要重新执行一次该命令，这时应该可以看到浏览器正常运行了。

问题：

1.这里可以看到，每改一个scripts都要手动执行编译是很不方便的。live-server~preview虽然自动刷新浏览器，但除非也能自动执行babel命令，否则得到的仍然是旧文件。

2.前面提到我们要配合webpack使用@babel/runtime而不是直接使用@babel/polyfill，这是由于如果不引入webpack，会提示你require找不到。

这2个问题，我们都在打包器中去解决。

# 打包器-问题

- 思考一个中型SPA应用，用到了50个第三方包，自己写的组件也有100个。有2个很直接的问题：

1. 如何组织这100个组件，都放在同一个JS文件吗，如果分散到多个文件中，又涉及到引用问题，通过<script>引用吗?
2. 如何引入这些第三方包，在HTML文件中插入50个<script>标签吗?

# 打包器-答案

- 第1个问题：JS的模块规范已经解决了此问题，webpack实现了ES6 module规范，因此用webpack即可解决；
- 第2个问题：打包器，将这50个第三方包都打包成vendor.js，以及这100个组件都打包成bundle.js。<script>只会引用vendor.js和bundle.js，不论增加多少第三包和组件都是如此。webpack作为打包器，其核心功能便是解决此问题。
- 不论是使用webpack本身，还是写组件，都用到了模块规范，因此先系统地介绍模块规范，再谈打包器本身。

备注：有某些第三方包，要求独占<script>，不与打包器一起，这类特殊情况不在本课件讨论范围

# 打包器-JS模块规范

- 对于初学者而言，JS模块规范看起来很凌乱，就一个模块引入功能，就有5种规范。。。这个混乱是由于服务端的JS与浏览端的JS的模块管理不一致导致的。服务端可同步或异步加载其他模块，浏览器端则必须异步加载模块。到了今天，这问题已经得到解决，借助于webpack，可以在服务端与浏览器都使用同样的模块规范。ES6 module是目前最新的模块规范，我们写应用代码时也是使用该规范，然而，写webpack配置时仍然用CommonJs规范，因为webpack是由node执行的，node也需要配置才能支持ES6 module。为了让大家在查资料时不至于十分混乱，简单介绍下不同的模块规范。

《JavaScript 模块化七日谈》：https://huangxuan.me/2015/07/09/js-module-7day/

《JS模块规范：AMD、UMD、CMD、CommonJS、ES6 Module》：https://segmentfault.com/a/1190000012419990

# 打包器-JS模块规范

- CommonJS(通过require函数引用模块，nodejs使用，webpack是通过nodejs执行的，所以要用该规范)
- AMD与RequireJS（RequireJS是AMD规范的实现）
- CMD与SeaJs（SeaJS是CMD规范的实现）
- UMD（兼容AMD和CMD）
- ES6 Module(应用统一使用该规范，通过import引用模块，通过export导出模块)

# 打包器-webpack介绍

- 在讨论Webpack之前，务必先做完“起步”的练习，获得一个感性的了解。 https://www.webpackjs.com/guides/getting-started/
- Webpack的概念：

1. 入口(entry)：指明源文件路径
2. 输出(ouput)：指明输出的文件路径
3. loader：对不同类型文件执行处理，比如调用babel对JS编译
4. 插件(plugins)：更大范围的任务，比如压缩、混淆代码

# 打包器-webpack用法

- webpack的用法非常多，本课重点是演示打包器的核心功能——打包，其他工程化的功能，像压缩，混淆，也是通过webpack完成，属于下节课内容。下面通过示例演示它的打包功能。

# 打包器~用法示例

## 对上周作业做一下改造

1.为便于后续讨论，将scripts/改名为src/,并将src/app.js改为src/index.js

2.修改src/index.js，在开头添加：

```
import "@babel/polyfill";
import React from 'react';
import ReactDOM from 'react-dom';
import axios from 'axios';
```

说明:根据ES6 Module规范引入模块，不用再写<script>，webpack会自动处理

3.修改index.html，删除所有的JS引用，只引用一会要生成的dist/bundle.js：

```
<body>
    <div id="app"></div>
    <script src="dist/bundle.js">
    </script>
</body>
```

说明:webpack默认将src/index.js连同它引用的模块都打包输出为dist/bundle.js，所以只引用该js文件即可。

# 打包器~用法示例

## 3.创建webpack.config.js

```
const path = require('path');

module.exports = {
  entry: './src/index.js',
  output: {
    filename: 'bundle.js',
    path: path.resolve(__dirname, 'dist')
  },
  module: {
    rules: [
      { test: /\.js$/, use: 'babel-loader'}
    ]
  }
};
```

说明：

entry：配置输入为src/index.js

output：配置输出为dist/bundle.js

module：配置loader，如果检测到js文件，就该文件做babel翻译，babel-loader将根据.babelrc去处理。

## 4.执行webpack命令

```
$npx webpack
```

```
Hash: d67e74623aa618c91d39
Version: webpack 4.41.2
Time: 4411ms
Built at: 2019-11-11 01:03:58
    Asset      Size  Chunks             Chunk Names
bundle.js  235 KiB       0  [emitted]  main
Entrypoint main = bundle.js
[134] ./src/index.js 13.2 KiB {0} [built]
    + 341 hidden modules
```

看到这样的输出表示生成OK了，将有dist/bundle.js文件，这时打开浏览器应该一切正常。

# 打包器~解决编译器一节发现的问题

问题1：每次修改原始文件，都要手动执行babel,现在是要手动执行webpack?

解决方法：webpack提供了2种方案解决此问题。

1. webpack --watch

2.webpack-dev-server

第1种方法监听到文件修改就会自动打包，可与VScode的live-server-preview配合

第2种方法则是自己启动web服务器并做实时重打包。

实际工程要求使用第2种。

问题2：怎么将@babel/polyfill换成@babel/runtime?

1.安装@babel/runtime

```
$npm install @babel/runtime
```

2.删除src/index.js中对@babel/polyfill的引入

```
--- a/demo2/src/index.js
+++ b/demo2/src/index.js
@@ -1,4 +1,3 @@
-import "@babel/polyfill";
 import React from 'react';
```

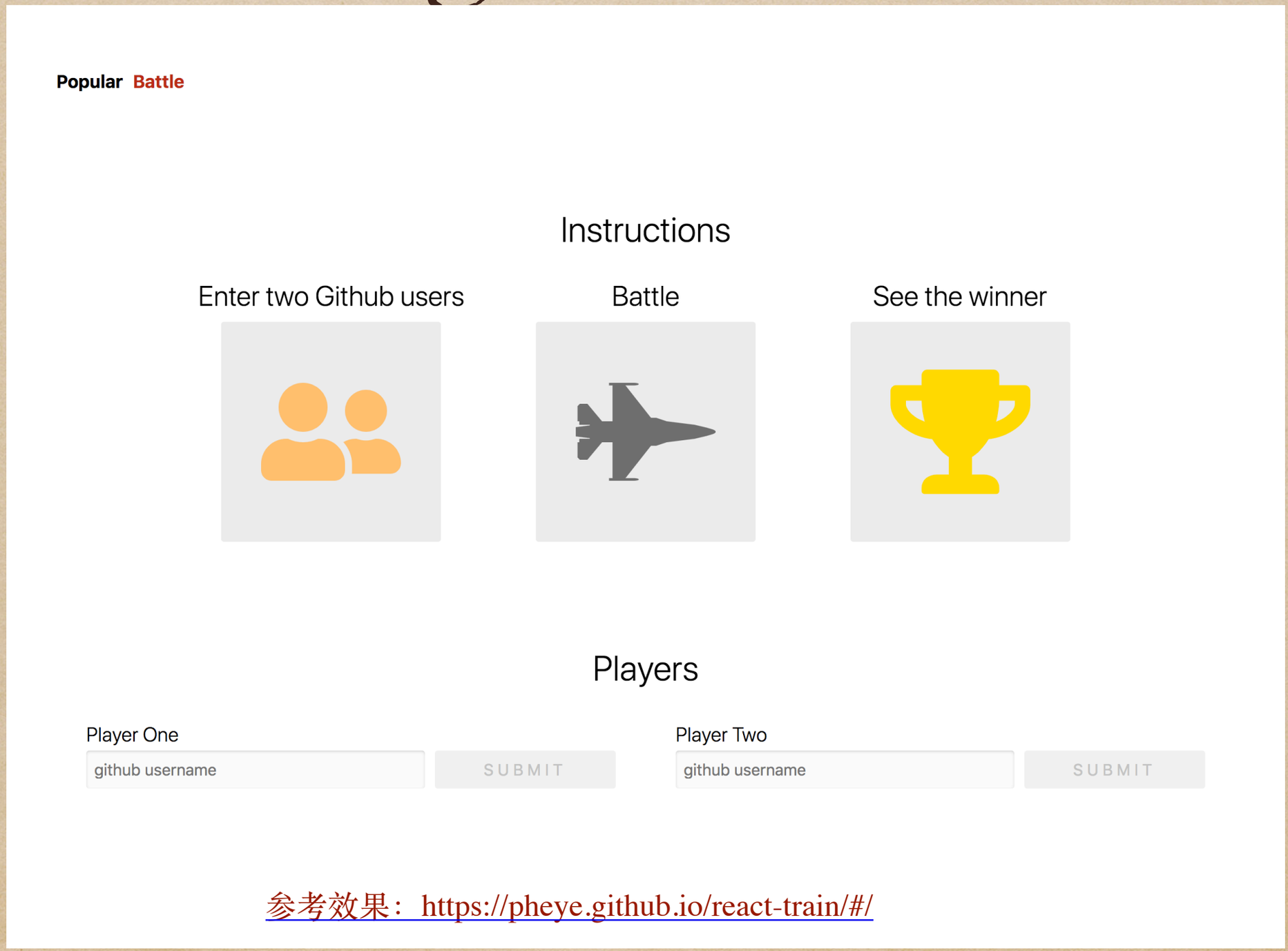
3.修改.babelrc,删除preset-env的选项，引入plugin-transform-runtime插件：

```
{
    "presets": [
        "@babel/preset-env",
        "@babel/preset-react"
    ],
    "plugins": [
        "@babel/plugin-transform-runtime",
        "@babel/plugin-proposal-class-properties"
    ]
}
```

# 作业：完成github热门项目工程化

- 完成前面示例对作业的工程化
- 集成webpack-dev-server
- 使用@babel/runtime而不是@babel/polyfill
- 思考CSS文件如何引入,font-awesome是否也能import?

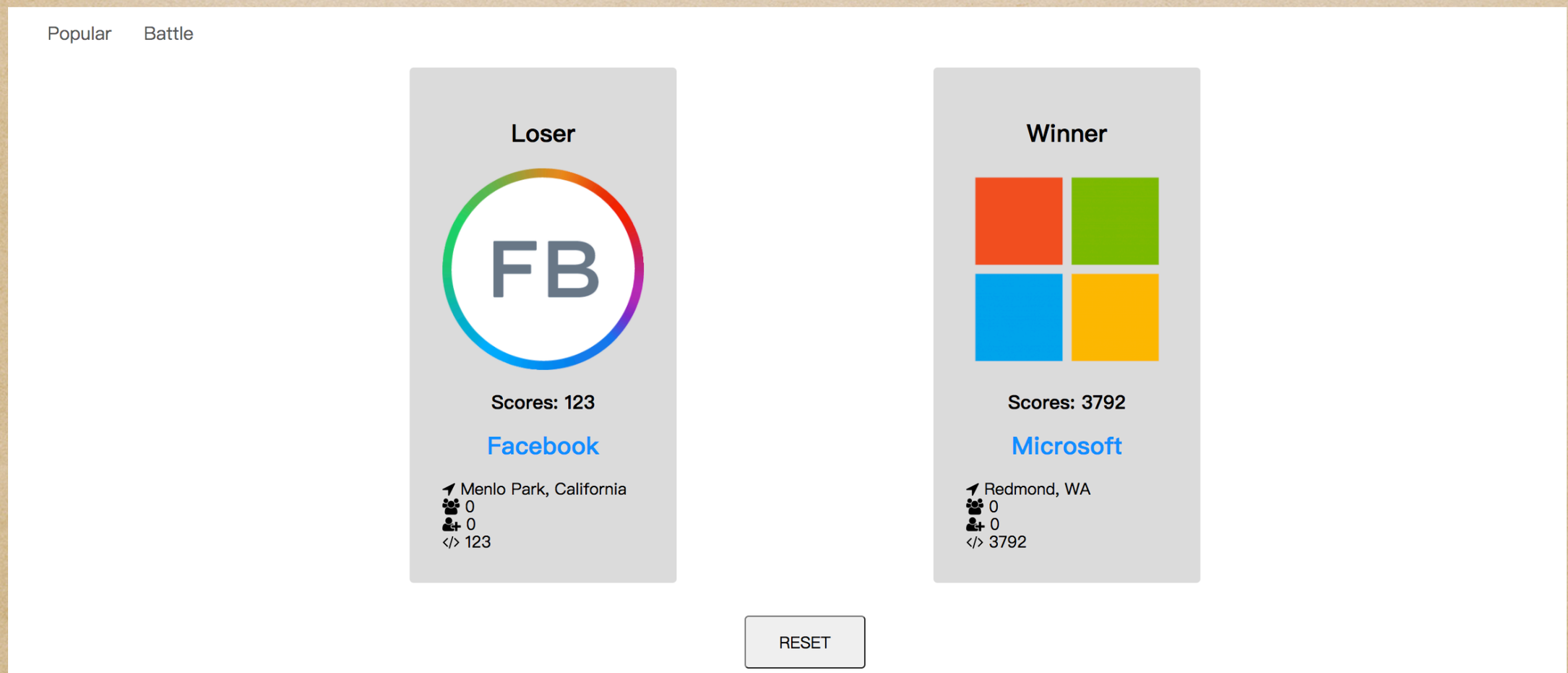
# 作业：增强github热门项目功能




参考效果：https://pheye.github.io/react-train/#/

# 作业：增强github热门项目功能

## 点击Battle后的结果页（要求刷新保持）




参考效果：https://pheye.github.io/react-train/#/

# 作业：增强github热门项目功能

- 实现Battle页面
- 导航栏的Popular和Battle，点击后切换页面
- Popular页面给作业增加无限加载功能。即目前只显示30个项目，下拉到底时应该加载第2页。必须配合第3方插件实现。参考插件：react-infinite-scroller。
- 使用Formik验证表单的有效性
- 响应式必须处理