

# Web dev

Joshua Yu

29 Aug. 2018

## 目录

### 1 Project feature request and problems

### 2 Things to be done before commit

### 3 Caveats

### 4 Git

|       |                                    |  |
|-------|------------------------------------|--|
| 4.1   | Basic Git Workflow . . . . .       |  |
| 4.2   | How to Backtrack in Git . . . . .  |  |
| 4.2.1 | head commit . . . . .              |  |
| 4.2.2 | git checkout . . . . .             |  |
| 4.2.3 | git reset I . . . . .              |  |
| 4.2.4 | git reset II . . . . .             |  |
| 4.3   | Git Branching . . . . .            |  |
| 4.3.1 | git merge . . . . .                |  |
| 4.3.2 | delete branch . . . . .            |  |
| 4.3.3 | Conclusion . . . . .               |  |
| 4.4   | Git Teamwork . . . . .             |  |
| 4.4.1 | git clone . . . . .                |  |
| 4.4.2 | git remote -v . . . . .            |  |
| 4.4.3 | git fetch . . . . .                |  |
| 4.4.4 | git merge . . . . .                |  |
| 4.4.5 | git push . . . . .                 |  |
| 4.5   | create-react-app and git . . . . . |  |

4.6 Your branch and 'origin/master' have diverged, how to undiverge branches?

## 5 VSCode

## 6 React

|       |   |
|-------|---|
| 6.1   | 创建React项目 Add React to a New Application . . . . .      |
| 6.2   | JSX . . . . .   |
| 6.2.1 | JSX caveats . . . . .                                   |
| 6.2.2 | JSX conditionals . . . . .                              |
| 6.2.3 | .map in JSX . . . . .                                   |
| 6.2.4 | Keys in JSX . . . . .                                   |
| 6.3   | The component . . . . .                                 |
| 6.4   | Components and advanced JSX . . . . .                   |
| 6.4.1 | Put Logic in a Render Function . . . . .                |
| 6.4.2 | Use this in a Component . . . . .                       |
| 6.4.3 | Use an Event Listener in a Component . . . . .          |
| 6.5   | Components render other components . . . . .            |
| 6.5.1 | A Component in a Render Function . . . . .              |
| 6.6   | this.props . . . . .                                    |
| 6.6.1 | Access a Component's props . . . . .                    |
| 6.6.2 | Pass 'props' to a Component . . . . .                   |
| 6.6.3 | Render a Component's props . . . . .                    |
| 6.6.4 | Pass props From Component To Component . . . . .        |
| 6.6.5 | Receive an Event Handler as a prop . . . . .            |
| 6.6.6 | this.props.children . . . . .                           |
| 6.6.7 | defaultProps . . . . .                                  |
| 6.7   | this.state . . . . .                                    |
| 6.7.1 | Setting Initial State . . . . .                         |
| 6.7.2 | Update state with this.setState . . . . .               |
| 6.7.3 | Call this.setState from Another Function . . . . .      |
| 6.8   | Child Components Update Their Parents' state . . . . .  |
| 6.9   | Child Components Update Their Siblings' props . . . . . |
| 6.10  | A controlled component example . . . . .                |
| 6.11  | Advanced React Techniques . . . . .                     |

|           |  |       |
|-----------|--|-------|
| 6.11.1    | React styles                                 | ..... |
| 6.11.2    | Another React pattern                        | ..... |
| 6.11.3    | propTypes                                    | ..... |
| 6.11.4    | PropTypes in Stateless Functional Components | ..... |
| 6.11.5    | React forms                                  | ..... |
| 6.12      | Lifecycle Methods                            | ..... |
| 6.12.1    | Mounting lifecycle methods                   | ..... |
| 6.12.2    | Updating/unmounting lifecycle methods        | ..... |
| <b>7</b>  | <b>react</b>                                 |       |
| 7.1       | react this 以下都为没在constructor里面进行this绑定       | ..... |
| 7.2       | react styles                                 | ..... |
| 7.3       | less   | ..... |
| 7.4       | Array map method                             | ..... |
| 7.5       | react高阶组件 (high order component)             | ..... |
| <b>8</b>  | <b>wechat mini program</b>                   |       |
| 8.1       | 框架   | ..... |
| <b>9</b>  | <b>JavaScript</b>                            |       |
| 9.1       | js扩展运算符(spread)是三个点(...)                     | ..... |
| 9.2       | object                                       | ..... |
| 9.2.1     | Looping Through Objects                      | ..... |
| 9.3       | class  | ..... |
| 9.4       | Request                                      | ..... |
| 9.5       | Errors                                       | ..... |
| 9.6       | Promise                                      | ..... |
| 9.7       | Async await                                  | ..... |
| 9.7.1     | The async Keyword                            | ..... |
| 9.7.2     | The await Operator                           | ..... |
| <b>10</b> | <b>CSS</b>                                   |       |
| 10.1      | Caveats                                      | ..... |
| 10.2      | Media Queries                                | ..... |
| 10.3      | CSS Display and Positioning                  | ..... |

|  |  |
|--|--|
| 10.4 The box model . . . . .                   |  |
| 10.5 Non-Inherited properties . . . . .        |  |
| 10.6 Learn Responsive Design . . . . .         |  |
| 10.6.1 Percentages: Padding & Margin . . . . . |  |
| 10.7 display property . . . . .                |  |
| 10.8 Grid layout . . . . .                     |  |
| 10.9 p tag . . . . .                           |  |

## 11 Node

## 12 npm (or yarn)

## 13 DOM

## 14 cmd

## 15 中文笔记

## 16 命名规范 Name convention

## 17 Abbr

# 1 Project feature request and problems

1. 怎样知道priceArr的数据结构 (E:\eastSun\ylx-pc-beta\src\components\LowPriceCalendar\index.js)

```
priceArr
Array(0) [, ...]
[[StableObjectId]]: 6
2019-02-02: Object {price: 57, surplusStock: 99}
2019-02-09: Object {price: 57, surplusStock: 99}
2019-02-16: Object {price: 57, surplusStock: 99}
2019-02-23: Object {price: 57, surplusStock: 99}
2019-03-02: Object {price: 57, surplusStock: 99}
2019-03-09: Object {price: 57, surplusStock: 99}
2019-03-16: Object {price: 57, surplusStock: 99}
```

```
2019-03-23: Object {price: 57, surplusStock: 99}
2019-03-30: Object {price: 57, surplusStock: 99}
length: 0
__proto__: Array(0) [, ...]
```

2. 与后台进行数据交互时，注意判断是字符串还是数字，比如 `needEmail === "1"`。  
“0” 也为 `true`
3. 使用 `localStorage` 的时候，打开一个标签然后另外再打开一个标签，则第一个标签里面的内容会变成第二个标签的内容
4. 微信门票列表搜索框，当需要对历史搜索里面的数据进行修改时，点击输入框后文字消失

## 2 Things to be done before commit

1. Indent using tabs, and set tab size to 2
2. Delete `console.log()`
3. Format document by pressing `shift+alt+f` in VSCode

## 3 Caveats

1. 以后需要避免测试人员回家了，我还有bug没有解决
2. 门票提交订单右半部分高度不随左半部分出游人增多而增加，可以新增一个div
3. codecademy 里面少一个标点符号或者拼写错误都会导致编译不通过
4. 找错的时候，首先应该用chrome developer tool确定位置

```
5. let tourists = 'John,'
   tourists.split(',').map(item => ({name: item }))
```

will return an array containing two objects

```
[{name: 'John'}, {name: ''}]
```

6. 如果当前页面有从localStorage里面取数据， 那么就不能直接在url上面写参数 (比如知道某个goodsTicketId， 然后直接加到url后面)， 因为请求的数据会跟从localStorage面取的数据不一样导致混淆
7. dangerouslySetInnerHTML 里面的样式是用原生css写
8. 注意将less文件导入到js文件中
9. chrome和firefox浏览器之间可能有兼容性问题， 比如用css属性设置文字超过四行用省略号
10. When debugging in VSCode, if you want a variable live updating then add it to `watch`, there's no live updating in the `debug console`
11. 数据在loading完成之前是undefined， 所以不能对其进行诸如 `address.constructor === Object` 的操作 (不能读取其constructor property)
12. `fontSize`比`lineHeight`应该小1
13. When a breakpoint is not hit in VSCode, you may need to stop by pressing `shift+F5` and then restart by pressing `F5`, and also if there's a new browser tab open then the breakpoint won't be hit. You can try copy the path url into a new tab and see
14. 使用iconfont的时候， 不知道怎么引用到项目中， 可以点击“下载至本地” 里面包含一个使用demo
15. You can set a breakpoint in VSCode, then add a Watch in chrome, because chrome has auto-complete feature. The debug config file `launch.json`'s url needs to be set according to the running port
16. minimize window by shortcut: `alt+space+n`
17. 用chrome来调试的时候， 可能需要用到`_this`来获取一些变量或对象等的值
18. Postman需要开启全局代理才能请求到数据
19. chrome inspect: margin 红 padding 绿 content 蓝
20. Maybe you need to set the breakpoint inside the body of an arrow function in order to be hit

21. After configured webpack.config.js file, you may need to recompile to make it work
22. Chrome developer tools: open the **Elements** tab to inspect the compiled html
23. **import** vs **require**: import can bring in only the desired function

```
import {countItems} from 'math_array_functions'
```

24. `<meta name="viewport" content="width= device-width, initial-scale= 1">`
25. FTP is the primary method by which people upload files to web servers. However, services like GoDaddy and others often try to provide simpler ways to upload files.
26. There is a space between **body** and **asterisk**

```
body, body *{  
    margin: unset;  
    padding: unset;  
}
```

27. Sometimes you have to close the page and then reload from webstorm to make the change take place

## 4 Git

### 4.1 Basic Git Workflow

`git init`

`git status` inspects the contents of the working directory and staging area

`git add filename_1 filename_2` add files/changes to the staging area. `git add .` or `git add -A` shortcut for adding all the files to the staging area

`git diff filename` press `q` on your keyboard to exit diff mode. If no content has been changed then nothing will appear. Diff between *add* and changes after *add*

`git commit -m "Complete first line of dialogue"` Standard Conventions for Commit Messages:

- Must be in quotation marks

- Written in the present tense
- Should be brief (50 characters or less) when using `-m`

`git log` Commits are stored chronologically in the repository and can be viewed using this command

## 4.2 How to Backtrack in Git

### 4.2.1 head commit

In Git, the commit you are currently on is known as the **HEAD** commit. In many cases, the most recently made commit is the **HEAD** commit

To see the **HEAD** commit, enter: `git show HEAD`

The output of this command will display everything the `git log` command displays for the **HEAD** commit, plus all the file changes that were committed

### 4.2.2 git checkout

What if you decide to change the ghost's line in the working directory, but then decide you wanted to discard that change?

`git checkout HEAD filename` or `git checkout -- filename` will restore the file in your working directory to look exactly as it did when you last made a commit

Here, `filename` again is the actual name of the file. If the file is named `changes.txt`, the command would be

```
git checkout HEAD changes.txt
```

### 4.2.3 git reset I

What if, before you commit, you accidentally delete an important line from `scene-2.txt`? Unthinkingly, you add `scene-2.txt` to the staging area. The file change is unrelated to the Larry/Laertes swap and you don't want to include it in the commit

We can unstage that file from the staging area using `git reset HEAD filename`

### 4.2.4 git reset II

Git enables you to rewind to the part before you made the wrong turn. You can do this with:



```
git reset commit_SHA
```

This command works by using the first 7 characters of the SHA of a previous commit. For example, if the SHA of the previous commit is 5d692065cf51a2f50ea8e7b19b5a7ae512f633ba, use:

```
git reset 5d69206
```

HEAD is now set to that previous commit

**Before reset:** HEAD is at the most recent commit

**After resetting:** HEAD goes to a previously made commit of your choice. You have in essence rewound the project's history

**Then** you may want to discard all the changes in that commit with

```
git checkout HEAD filename
```

## 4.3 Git Branching

Up to this point, you've worked in a single Git branch called **master**. Git allows us to create branches to experiment with versions of a project. Imagine you want to create version of a story with a happy ending. You can create a new branch and make the happy ending changes to that branch only. It will have no effect on the **master** branch until you're ready to merge the happy ending to the master branch

- You can use following command to answer the question: “which branch am I on?” `git branch`
- To create a new branch, use: `git branch new_branch`. Also, branch names can't contain whitespaces: `new-branch` and `new_branch` are valid branch names, but `new branch` is not
- You can switch to the new branch with: `git checkout branch_name` (notice the \* is now over the new branch)
- Once you switch branch, you now able to make commits on the branch that have no impact on **master**

### 4.3.1 git merge

- Switch to master branch with `git checkout master`
- Merge new branch to master branch using `git merge new_branch`

- The merge is a "fast forward" because Git recognizes that **fencing** (created new branch) contains the most recent commit. Git fast forwards **master** to be up to date with **fencing**

#### 4.3.2 delete branch

In Git, branches are usually a means to an end. You create them to work on a new project feature, but the end goal is to merge that feature into the master branch. After the branch has been integrated into master, it has served its purpose and can be deleted

The command `git branch -d branch_name` will delete the specified branch from your Git project. If some feature branches were never merged into **master**, then use the uppercase D, like `git branch -D branch_name`

#### 4.3.3 Conclusion

- `git branch`: Lists all a Git project's branches
- `git branch branch_name`: Creates a new branch
- `git checkout branch_name`: Used to switch from one branch to another
- `git merge branch_name`: Used to join file changes from one branch to another
- `git branch -d branch_name`: Deletes the branch specified

### 4.4 Git Teamwork

#### 4.4.1 git clone

```
git clone remote_location clone_name
```

- **remote\_location** tells Git where to go to find the remote. This could be a web address, or a filepath
- **clone\_name** is the name you give to the directory in which Git will clone the repository

For example, you're collaborating with Sally, the Git remote Sally started is called: **science-quizzes**

```
git clone science-quizzes my-quizzes
```

my-quizzes is your local copy of the science-quizzes Git project. If you commit changes to the project here, Sally will not know about them

#### 4.4.2 git remote -v

Nice work! We have a clone of Sally's remote on our computer. One thing that Git does behind the scenes when you clone **science-quizzes** is give the remote address the name **origin**, so that you can refer to it more conveniently. In this case, Sally's remote is **origin**.

You can see a list of a Git project's remotes with the command: `git remote -v`. The remote is listed twice: once for (**fetch**) and once for (**push**). We'll learn about these later in the lesson

#### 4.4.3 git fetch

An easy way to see if changes have been made to the remote and bring the changes down to your local copy is with:

```
git fetch
```

This command will not *merge* changes from the remote into your local repository. It brings those changes onto what's called a *remote branch*. Learn more about how this works below

#### 4.4.4 git merge

Even though Sally's new commits have been fetched to your local copy of the Git project, those commits are on the **origin/master** branch. Your *local master* branch has not been updated yet, so you can't view or make changes to any of the work she has added

In *Lesson III, Git Branching* we learned how to merge branches. Now we'll use the `git merge` command to integrate **origin/master** into your local **master** branch. The command:

```
git merge origin/master | git merge origin/dev
```

 will accomplish this for us

#### 4.4.5 git push

Now it's time to share our work with Sally.

```
git push origin your_branch_name | git push origin HEAD:dev
```

will push your branch up to the remote, `origin`. From there, Sally can review your branch and merge your work into the master branch, making it part of the definitive project version.

#### 4.5 create-react-app and git

1. `create-react-app myapp` will initialize a git repository
2. create a new repository on github, then push existing repository from command line using following commands

```
git remote add origin https://github.com/Jiapan-Yu/newTest.git
git push -u origin master
```
3. then create a new branch with `git branch dev` and switch to that branch

```
git checkout dev
```

#### 4.6 Your branch and 'origin/master' have diverged, how to undiverge branches?

You can review the differences with a: `git log HEAD..origin/master` before pulling it (fetch + merge), see [merge or rebase](#)

### 5 VSCode

- 
- source control: modified 为蓝色 added 为暗橙色
- 搜索只能搜内容，无法搜索哪个文件夹
- type `tab` key twice to autocomplete

## 6 React

### 6.1 创建React项目 [Add React to a New Application](#)

1. `npm install -g create-react-app`
2. `create-react-app myapp`
3. `cd myapp`
4. `npm start`
5. When you're ready to deploy to production, running `npm run build` will create an optimized build of your app in the `build` folder

```
import React from 'react';
import ReactDOM from 'react-dom';

class App extends React.Component {
  render() {
    return (<h1>React setup</h1>);
  }
}

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

6. 默认所有配置是隐藏起来的，要想自定义配置，需要运行一个命令：`npm run eject`。接着`package.json`里面的`scripts`键会变为`node`。`npm start` 运行正常
7. install `less` and `mini-css-extract-plugin` to use `less` and extract `css` into one file, configured `webpack.config.dev.js`. Add following lines in the correct places

```
const MiniCssExtractPlugin = require("mini-css-extract-plugin");

,
```

```
// adds support for css and less
{
  test: /\.(css|less)$/, /* need this less */
  use: [
    MiniCssExtractPlugin.loader,
    "css-loader",
    "less-loader" /* need this line */
  ]
}

new MiniCssExtractPlugin({
  // Options similar to the same options in webpackOptions.output
  // both options are optional
  filename: "[name].css",
  chunkFilename: "[id].css"
}),
```

## 6.2 JSX

### 1. Event Listeners in JSX

```
<img onClick={myFunc} />
```

An event listener attribute's name should be something like `onClick` or `onMouseOver`: the word `on`, plus the type of event that you're listening for. [Supported Events](#)

**Note** that in HTML, event listener names are written in all lowercase, such as `onclick` or `onmouseover`. In JSX, event listener names are written in camelCase, such as `onClick` or `onMouseOver`

### 2. Variable Attributes in JSX

```
// Use a variable to set the 'height' and 'width' attributes:
```

```
const sideLength = "200px";
```

```
const panda = (
```

```

);
```

3. Everything inside of the curly braces will be treated as regular JavaScript

```
ReactDOM.render(
  <h1>{2 + 3}</h1>,
  document.getElementById('app')
);
```

The result is 5

4. ReactDOM.render() is the most common way to render JSX:

```
ReactDOM.render(<h1>Hello world</h1>, document.getElementById('app'));
```

5. There's a rule that we haven't mentioned: a JSX expression must have exactly one outermost element.
6. If a JSX expression takes up more than one line, then you must wrap the multi-line JSX expression in parentheses. This looks strange at first, but you get used to it:

```
const theExample = (
  <a href="https://www.example.com">
    <h1>
      Click me!
    </h1>
  </a>
);
```

7. JSX elements are treated as JavaScript expressions. They can go anywhere that JavaScript expressions can go.

That means that a JSX element can be saved in a variable, passed to a function, stored in an object or array...you name it

8. JSX is a syntax extension for JavaScript. It was written to be used with React. JSX code looks a lot like HTML

### 6.2.1 JSX caveats

1. You can't use JSX until you've imported react
2. One outermost tag
3. You can not inject an if statement into a JSX expression.

This code will break:

```
(  
  <h1>  
    {  
      if (purchase.complete) {  
        'Thank you for placing an order!'  
      }  
    }  
  </h1>  
)
```

4. In JSX, self-closing tags have to include the slash, otherwise it will raise an error
5. In JSX, you can't use the word class! You have to use className instead:

```
<h1 className="big">Hey</h1>
```

This is because JSX gets translated into JavaScript, and `class` is a reserved word in JavaScript

### 6.2.2 JSX conditionals

1. The Ternary Operator



```
const headline = (
  <h1>
    { age >= drinkingAge ? 'Buy Drink' : 'Do Teen Stuff' }
  </h1>
);
```

## 2. &&

```
const tasty = (
  <ul>
    <li>Applesauce</li>
    { !baby && <li>Pizza</li> }
    { age > 15 && <li>Brussels Sprouts</li> }
    { age > 20 && <li>Oysters</li> }
    { age > 25 && <li>Grappa</li> }
  </ul>
);
```

### 6.2.3 .map in JSX

If you want to create a list of JSX elements, then `.map()` is often your best bet. It can look odd at first:

```
const strings = ['Home', 'Shop', 'About Me'];

const listItems = strings.map(string => <li>{string}</li>);

<ul>{listItems}</ul>
```

### 6.2.4 Keys in JSX

A **key** is a JSX attribute. The attribute's name is **key**. The attribute's value should be something unique, similar to an `id` attribute

```
const people = ['Rowe', 'Prevost', 'Gare'];

const peopleLis = people.map((person, i) =>
```

```
// expression goes here:  
<li key={'person_' + i}>{person}</li>  
);
```

### 6.3 The component

1. React applications are made out of *components*. A component is a small, reusable chunk of code that is responsible for one job. That job is often to render some HTML.

```
import React from 'react';  
import ReactDOM from 'react-dom';  
  
class MyComponentClass extends React.Component {  
  render() {  
    return <h1>Hello world</h1>;  
  }  
};  
  
ReactDOM.render(  
  <MyComponentClass />,  
  document.getElementById('app')  
);
```

2. For now, just know that you get the React library via `import React from 'react';`
3. To clarify: the DOM is *used* in React applications, but it isn't *part* of React  
`import ReactDOM from 'react-dom';`
4. By subclassing `React.Component`, you create a new component class. This is not a component! A component class is more like a factory that produces components. When you start making components, each one will come from a component class
5. **The Render Function:** All you know so far is that its name is `render`, it needs a return statement for some reason, and you have to include it in the body of your component class declaration

## 6. Create a Component Instance:

```
<MyComponentClass />
```

7. **Render A Component:** `<MyComponentClass />` will call its render method, which will return the JSX element `<h1>Hello world</h1>`. `ReactDOM.render()` will then take that resulting JSX element, and add it to the virtual DOM. This will make "Hello world" appear on the screen.

## 6.4 Components and advanced JSX

### 6.4.1 Put Logic in a Render Function

```
class Random extends React.Component {  
  render() {  
    const n = Math.floor(Math.random() * 10 + 1);  
    return <h1>The number is {n}!</h1>;  
  }  
}
```

### 6.4.2 Use this in a Component

```
class IceCreamGuy extends React.Component {  
  get food() {  
    return 'ice cream';  
  }  
  
  render() {  
    return <h1>I like {this.food}</h1>;  
  }  
}
```

### 6.4.3 Use an Event Listener in a Component

```
render() {  
  return (  
    <div onMouse={myFunc}>  
      </div>
```

```
);  
}
```

Recall that an event handler is a function that gets called in response to an event. In the above example, the event handler is `myFunc()`.

In React, you define event handlers (functions) as methods on a component class. Like this:

```
class MyClass extends React.Component {  
  myFunc() {  
    alert('Stop it. Stop hovering.');  }  
  
  render() {  
    return (  
      <div onMouse={this.myFunc}>  
      </div>  
    );  
  }  
}
```

Almost all functions that you define in React will be defined in this way, as methods in a class

## 6.5 Components render other components

### 6.5.1 A Component in a Render Function

1. When you use named exports, you always need to wrap your imported names in curly braces, such as:

```
import { faveManifestos, alsoRan } from './Manifestos';
```

2. To import a variable, you can use an `import` statement:

```
import { NavBar } from './NavBar.js';
```

you can omit the `.js` extension

3. Render methods can also return another kind of JSX: component instances

```
class OMG extends React.Component {
  render() {
    return <h1>Whooaa!</h1>;
  }
}
```

```
class Crazy extends React.Component {
  render() {
    return <OMG />;
  }
}
```

In the above example, **Crazy**'s render method **returns** an instance of the **OMG** component class. You could say that **Crazy** renders an `<OMG />`

## 6.6 this.props

Information that gets passed from one component to another is known as "props."

### 6.6.1 Access a Component's props

To see a component's props object, you use the expression `this.props`

```
class PropsDisplayer extends React.Component {
  render() {
    const stringProps = JSON.stringify(this.props);

    return (
      <div>
        <h1>CHECK OUT MY PROPS OBJECT</h1>
        <h2>{stringProps}</h2>
      </div>
    );
  }
}
```

```
ReactDOM.render(<PropsDisplayer />, document.getElementById('app'));
```

### 6.6.2 Pass 'props' to a Component

If you want to pass information that isn't a string, then wrap that information in curly braces

```
<Greeting myInfo={["top", "secret", "lol"]} />
```

```
<Greeting name="Frarthur" town="Flundon" age={2} haunted={false} />
```

### 6.6.3 Render a Component's props

```
class Greeting extends React.Component {  
  render() {  
    return <h1>Hi there, {this.props.firstName}!</h1>;  
  }  
}
```

```
ReactDOM.render(  
  <Greeting firstName='Joshua' />,  
  document.getElementById('app')  
)
```

### 6.6.4 Pass props From Component To Component

**A curmudgeonly clarification about grammar:** You may have noticed some loose usage of the words `prop` and `props`. `props` is the name of the object that stores passed-in information. `this.props` refers to that storage object. At the same time, each piece of passed-in information is called a `prop`. This means that `props` could refer to two pieces of passed-in information, or it could refer to the object that stores those pieces of information

#### Greeting.js

```
import React from 'react';  
  
export class Greeting extends React.Component {  
  render() {  
    return <h1>Hi there, {this.props.name}!</h1>;  
  }  
}
```

```
}  
}
```

### App.js

```
import React from 'react';  
import ReactDOM from 'react-dom';  
import {Greeting} from './Greeting';  
  
class App extends React.Component {  
  render() {  
    return (  
      <div>  
        <h1>  
          Hulloo and, "Welcome to The Newzz," "On Line!"  
        </h1>  
        <Greeting name="Joshua" />  
        <article>  
          Latest newzz:  where is my phone?  
        </article>  
      </div>  
    );  
  }  
}
```

```
ReactDOM.render(  
  <App />,  
  document.getElementById('app')  
)
```

#### 6.6.5 Receive an Event Handler as a prop

### Button.js

```
import React from 'react';
```

```

export class Button extends React.Component {
  render() {
    return (
      <button onClick={this.props.onClick}>
        Click me!
      </button>
    );
  }
}

```

### Talker.js

```

import React from 'react';
import ReactDOM from 'react-dom';
import { Button } from './Button';

class Talker extends React.Component {
  handleClick() {
    let speech = '';
    for (let i = 0; i < 10000; i++) {
      speech += 'blah ';
    }
    alert(speech);
  }

  render() {
    return <Button onClick={this.handleClick} />;
  }
}

ReactDOM.render(
  <Talker />,
  document.getElementById('app')
);

```



Great! You just passed a function from `<Talker />` to `<Button />`. `<Talker />` is the *parent* component class.

**Name confusion demystify:** `<Button />` is not an HTML-like JSX element; it's a component instance. Names like `onClick` only create event listeners if they're used on HTML-like JSX elements. Otherwise, they're just ordinary prop names.

#### 6.6.6 `this.props.children`

```
import { LilButton } from './LilButton';

class BigButton extends React.Component {
  render() {
    console.log(this.props.children);
    return <button>Yo I am big</button>;
  }
}
```

```
// Example 1
<BigButton>
  I am a child of BigButton.
</BigButton>
```

```
// Example 2
<BigButton>
  <LilButton />
</BigButton>
```

```
// Example 3
<BigButton />
```

In Example 1, `<BigButton>`'s `this.props.children` would equal the text, "I am a child of BigButton."

In Example 2, `<BigButton>`'s `this.props.children` would equal a `<LilButton />` component.

In Example 3, `<BigButton>`'s `this.props.children` would equal `undefined`.

If a component has more than one child between its JSX tags, then `this.props.children` will return those children in an array. However, if a component has only one child, then `this.props.children` will return the single child, not wrapped in an array.

#### 6.6.7 defaultProps

```
class Button extends React.Component {
  render() {
    return (
      <button>
        {this.props.text}
      </button>
    );
  }
}

// defaultProps goes here:
Button.defaultProps = { text: 'I am a button' };

ReactDOM.render(
  <Button />,
  document.getElementById('app')
);
```

### 6.7 this.state

Dynamic information is information that can change. There are two ways for a component to get dynamic information: **props** and **state**. Besides **props** and **state**, every value used in a component should always stay exactly the same

### 6.7.1 Setting Initial State

To make a component have **state**, give the component a **state** property. This property should be declared inside of a constructor method, like this

```
class Example extends React.Component {  
  constructor(props) {  
    super(props);  
    this.state = { mood: 'decent' };  
  }  
  
  render() {  
    return <div></div>;  
  }  
}
```

```
<Example />
```

`this.state` should be equal to an object, like in the example above. This object represents the initial "state" of any component instance

### 6.7.2 Update state with `this.setState`

`this.setState()` takes two arguments: an object that will update the component's state, and a callback. You basically never need the callback (the callback is executed after `render()` function)

### 6.7.3 Call `this.setState` from Another Function

```
class Mood extends React.Component {  
  constructor(props) {  
    super(props);  
    this.state = { mood: 'good' };  
    this.toggleMood = this.toggleMood.bind(this);  
  }  
  
  toggleMood() {
```

```

    const newMood = this.state.mood === 'good' ? 'bad' : 'good';
    this.setState({ mood: newMood });
  }

  render() {
    return (
      <div>
        <h1>I'm feeling {this.state.mood}!</h1>
        <button onClick={this.toggleMood}>
          Click Me
        </button>
      </div>
    );
  }
}

```

in React, whenever you define an event handler that uses `this`, you need to add `this.methodName = this.methodName.bind(this)` to your constructor function.

*Any time that you call `this.setState()`, `this.setState()` AUTOMATICALLY calls `.render()` as soon as the state has changed.*

Think of `this.setState()` as actually being two things: `this.setState()`, immediately followed by `.render()`. Set breakpoint in `render()` method in order to inspect the changes made by `this.setState()` in chrome

That is why you can't call `this.setState()` from inside of the `.render()` method! `this.setState()` *automatically* calls `.render()`. If `.render()` calls `this.setState()`, then an infinite loop is created

## 6.8 Child Components Update Their Parents' state

See javascript.pdf

## 6.9 Child Components Update Their Siblings' props

See codecademy, this pattern occurs in React all the time!

## 6.10 A controlled component example

```
class NameForm extends React.Component {
  constructor(props) {
    super(props);
    this.state = {value: ''};

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

  handleChange(event) {
    this.setState({value: event.target.value});
  }

  handleSubmit(event) {
    alert('A name was submitted: ' + this.state.value);
    event.preventDefault();
  }

  render() {
    return (
      <form onSubmit={this.handleSubmit}>
        <label>
          Name:
          <input type="text" value={this.state.value} onChange={this.handleChange} />
        </label>
        <input type="submit" value="Submit" />
      </form>
    );
  }
}

ReactDOM.render(
```

```
<NameForm />,
document.getElementById('root')
);
```

## 6.11 Advanced React Techniques

### 6.11.1 React styles

In regular JavaScript, style names are written in hyphenated-lowercase:

```
const styles = {
  'margin-top':      "20px",
  'background-color': "green"
};
```

In React, those same names are instead written in camelCase:

```
const styles = {
  marginTop:      20,
  backgroundColor: "green"
};
```

This has zero effect on style property values, only on style property names.

In React, if you write a style value as a number (meaning no quotation marks), then the unit "px" is assumed. The exception is `lineHeight` attribute, because it accepts number and length at the same time

If you want to use units other than "px," you can use a string: `{ fontSize: "2em" }`

### 6.11.2 Another React pattern

A presentational component will always get rendered by a container component

When you separate a container component from a presentational component, the *presentational* component will always end up like this: one `render()` function, and no other properties

```
import React from 'react';
```

```
export class GuineaPigs extends React.Component {
  render() {
```

```

    let src = this.props.src;
    return (
      <div>
        <h1>Cute Guinea Pigs</h1>
        <img src={src} />
      </div>
    );
  }
}

```

If you have a component class with nothing but a render function, then you can rewrite that component class in a very different way. Instead of using `React.Component`, you can write it as JavaScript function! A component class written as a function is called a *stateless functional component*

See [javascript.pdf](#) for an example

### 6.11.3 propTypes

```

import React from 'react';

export class MessageDisplay extends React.Component {
  render() {
    return <h1>{this.props.message}</h1>;
  }
}

// This propTypes object should have
// one property for each expected prop:
MessageDisplay.propTypes = {
  message: React.PropTypes.string
};

```

Notice that the value of `propTypes` is an object, not a function! Each property on the `propTypes` object is called a `propTypes`

```

Runner.propTypes = {

```

```
message: React.PropTypes.string.isRequired,
style: React.PropTypes.object.isRequired,
isMetric: React.PropTypes.bool.isRequired,
miles: React.PropTypes.number.isRequired,
milesToKM: React.PropTypes.func.isRequired,
races: React.PropTypes.array.isRequired
};
```

Runner has six propTypes! Look at each one. Note that `bool` and `func` are abbreviated, but all other datatypes are spelled normally.

#### 6.11.4 PropTypes in Stateless Functional Components

```
import React from 'react';

export const GuineaPigs = (props) => {
  let src = props.src;
  return (
    <div>
      <h1>Cute Guinea Pigs</h1>
      <img src={src} />
    </div>
  );
}
```

```
GuineaPigs.propTypes = {
  src: React.PropTypes.string.isRequired
};
```

#### 6.11.5 React forms

### 6.12 Lifecycle Methods

#### 6.12.1 Mounting lifecycle methods

```
componentWillMount(){} render(){} componentDidMount(){}
```



When a component renders for the first time, `componentWillMount` gets called right before `render`.

When a component renders for the first time, `componentDidMount` gets called right after the HTML from `render` has finished loading.

Mounting lifecycle events only execute the first time that a component renders.

If your React app uses AJAX to fetch initial data from an API, then `componentDidMount` is the place to make that AJAX call. More generally, `componentDidMount` is a good place to connect a React app to external applications, such as web APIs or JavaScript frameworks. `componentDidMount` is also the place to set timers using `setTimeout` or `setInterval`.

### 6.12.2 Updating/unmounting lifecycle methods

There are five updating lifecycle methods:

- `componentWillReceiveProps`
- `shouldComponentUpdate`
- `componentWillUpdate`
- `render`
- `componentDidUpdate`

Whenever a component instance updates, it automatically calls all five of these methods, in order.

`componentWillUnmount` is the only unmounting lifecycle method.

`componentWillUnmount` gets called right before a component is removed from the DOM. If a component initiates any methods that require cleanup, then `componentWillUnmount` is where you should put that cleanup

## 7 react

1. ES6 React 组件引用本地图片问题: require里只能写字符串, 不能写变量
- 2.

## 7.1 react this 以下都为没在constructor里面进行this绑定

可以用普通函数的地方都可以用箭头函数。另外箭头函数在某些情况下 (是否只传event还是有其它参数和是在class methods处还是在事件触发处) 可以替代bind

在React里面, 传参要看传的只是event还是含有 (event貌似是一定有的) 其它参数 (包括其它参数为空的情形)。当含有其它参数的时候, 有以下几种写法 (可以参考ylx-pc门票列表), 箭头函数当用于事件触发的地方时能替代bind, 当class methods使用的时候就不可替代bind

```
1. handleClick(item) {  
    this.props.handleClick(item)  
}
```

或

```
handleClick = (item) => {  
    this.props.handleClick(item)  
}
```

```
<div className="history" onClick={() => this.handleClick(item)}></div>
```

```
2. handleClick = (item) => {  
    this.props.handleClick(item)  
}
```

```
<div className="history" onClick={this.handleClick.bind(this, item)}></div>
```

下面一行代码不行

```
<div className="history" onClick={this.handleClick(item)}></div>
```

```
3. handleClick(item) {  
    this.props.handleClick(item)  
}
```

```
<div className="history" onClick={this.handleClick.bind(this, item)}></div>
```

```
4. handleClick = (item) => () => {  
    this.props.handleClick(item)
```

```
} //called currying
```

```
<div className="history" onClick={this.handleClick(item)}></div>
```

下面一行代码不行

```
<div className="history" onClick={this.handleClick.bind(this, item)}></div>
```

当只为event的时候，箭头函数当class methods使用的时候可以替代bind (见1)。但是当用于事件触发的地方时就不能替代bind (见2) (可以参考javascript.pdf Child Components Update Their Parents' state in React)。可以用普通函数的地方都可以用箭头函数 (见3)

```
1. handleChange = (e) => {  
    const name = e.target.value;  
    this.props.onChange(name);  
} // 必须用箭头函数，普通函数不行
```

```
<select id="great-names" onChange={this.handleChange}></select>
```

```
2. handleChange(e) {  
    const name = e.target.value;  
    this.props.onChange(name);  
}
```

```
<select id="great-names" onChange={this.handleChange.bind(this)}></select>
```

下面一行代码不行

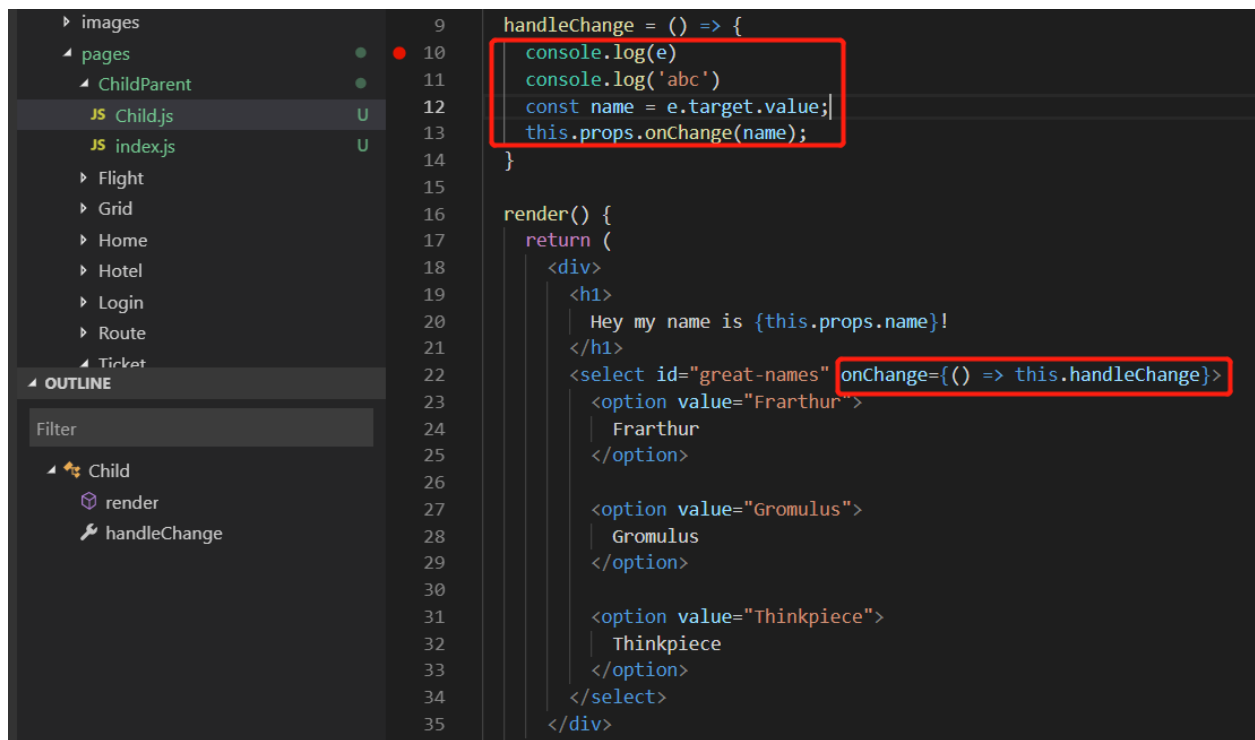
```
<select id="great-names" onChange={() => this.handleChange}></select>
```

```
3. handleChange = (e) => {  
    const name = e.target.value;  
    this.props.onChange(name);  
}
```

```
<select id="great-names" onChange={this.handleChange.bind(this)}></select>
```

下面一行代码不行

```
<select id="great-names" onChange={() => this.handleChange}></select>
```

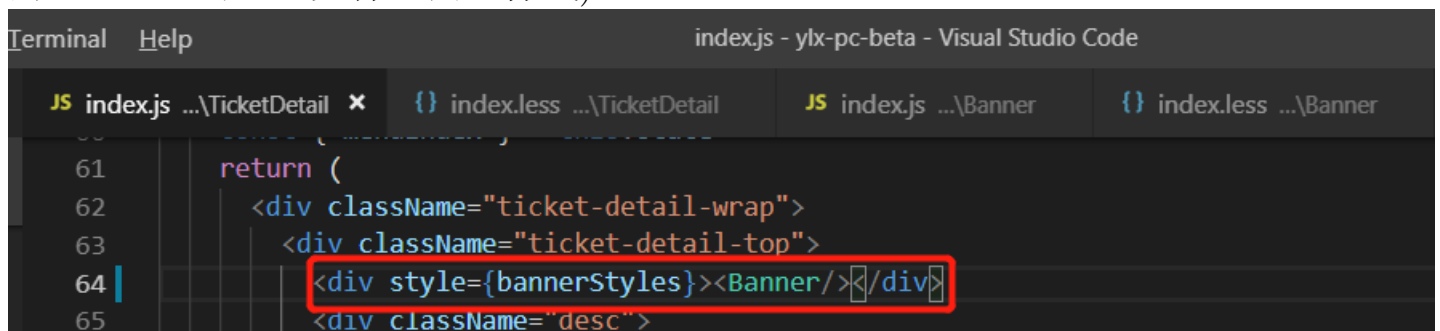


```
9  handleChange = () => {
10    console.log(e)
11    console.log('abc')
12    const name = e.target.value;
13    this.props.onChange(name);
14  }
15
16  render() {
17    return (
18      <div>
19        <h1>
20          Hey my name is {this.props.name}!
21        </h1>
22        <select id="great-names" onChange={this.handleChange}>
23          <option value="Frarthur">
24            Frarthur
25          </option>
26
27          <option value="Gromulus">
28            Gromulus
29          </option>
30
31          <option value="Thinkpiece">
32            Thinkpiece
33          </option>
34        </select>
35      </div>
```

The body of `handleChange` won't be executed, don't know why. 当传的是其它参数的时候 (图片中的情形), 在事件触发的地方调用class method需要加(), 否则对应的class method的body不会被执行

## 7.2 react styles

在项目中引用某个组件时, 如果想对其添加style, 则需要将那个组件包裹在div内, 接着在div上加style属性, 不能直接在那个组件上加style属性 (但是可以找到对应的className在less文件里面加样式)



```
61  return (
62    <div className="ticket-detail-wrap">
63      <div className="ticket-detail-top">
64        <div style={bannerStyles}><Banner/></div>
65      <div className="desc">
```

## 7.3 less

React + CSS Modules + LESS + Webpack 4 包含一个完整的less示例

## 7.4 Array map method

1.

2. Each child in an array or iterator should have a unique "key" prop

## 7.5 react高阶组件 (high order component)

- 高阶组件就是接受一个组件作为参数并返回一个新组件的函数
- 高阶组件是一个函数，并不是组件
- 尽量使用代理方式的高阶组件
- 

## 8 wechat mini program

### 8.1 框架

1. 开发者需要做的只是将页面的数据、方法、生命周期函数注册到框架中
2. 框架的核心是一个响应的数据绑定系统
3. 整个小程序框架系统分为两部分：视图层（View）和逻辑层（App Service）

小程序 服务号 订阅号 企业号

## 9 JavaScript

### 9.1 js扩展运算符(spread)是三个点(...)

作用：将一个数组转为用逗号分隔的参数序列

### 9.2 object

1. There are only seven fundamental data types in JavaScript, and six of those are the primitive data types: `string`, `number`, `boolean`, `null`, `undefined`, and

symbol. With the seventh type, `objects`, we open our code to more complex possibilities

2. Objects are *mutable* meaning we can update them after we create them!
3. You can delete a property from an object with the `delete` operator

```
const spaceship = {  
  'Fuel Type': 'Turbo Fuel',  
  homePlanet: 'Earth',  
  mission: 'Explore the universe'  
};
```

```
delete spaceship['Fuel Type'];
```

4. When the data stored on an object is a function we call that a *method*
5. Certain languages have privacy built-in for objects, but JavaScript does not have this feature. Rather, JavaScript developers follow naming conventions that signal to other developers how to interact with a property
6. `typeof()`: in JavaScript, the data type of null is an object. The return value of `typeof()` can be "string" "number" "boolean" "undefined" "object" "function"
7. JavaScript Types are Dynamic:

```
var x;           // Now x is undefined  
x = 5;           // Now x is a Number  
x = "John";      // Now x is a String
```

8. *avoid* using arrow functions when using `this` in a method
9. Property Value Shorthand:

factory function:

```
const monsterFactory = (name, age) => {  
  return {  
    name: name,  
    age: age
```

```
    }  
};
```

Imagine if we had to include more properties, that process would quickly become tedious! But we can use a destructuring technique, called *property value shorthand*, to save ourselves some keystrokes

```
const monsterFactory = (name, age) => {  
  return {  
    name,  
    age  
  }  
};
```

## 10. Destructured Assignment

```
const vampire = {  
  name: 'Dracula',  
  residence: 'Transylvania',  
  preferences: {  
    day: 'stay inside',  
    night: 'satisfy appetite'  
  }  
};
```

```
const { residence } = vampire;  
console.log(residence); // Prints 'Transylvania'
```

We can even use destructured assignment to grab nested properties of an object:

```
const { day } = vampire.preferences;  
console.log(day); // Prints 'stay inside'
```

## 11. Built-in Object Methods

object instance methods like: `.hasOwnProperty()`, `.valueOf()`

Object class methods such as `Object.assign()`, `Object.entries()`, and `Object.keys()`

### 9.2.1 Looping Through Objects

JavaScript has given us alternative solution for iterating through objects with the `for...in` syntax

## 9.3 class

1. Although the subclass automatically inherits the parent methods, you need to use the `super` keyword to set the parent properties
2. static methods: The `.now()` method is static, so you can call it directly from the class, but not from an instance of the class

```
static generateName() {  
    const names = ['Angel', 'Spike', 'Buffy', 'Willow', 'Tara'];  
    const randomNumber = Math.floor(Math.random()*5);  
    return names[randomNumber];  
}
```

3. In a `constructor()`, you must always call the `super` method before you can use the `this` keyword — if you do not, JavaScript will throw a reference error, it is best practice to call `super` on the first line of subclass constructors
4. When multiple classes share properties or methods, they become candidates for *inheritance* — a tool developers use to decrease the amount of code they need to write
5. Notice, we also prepended our property names with underscores (`_name` and `_behavior`), which indicate these properties should not be accessed directly

## 9.4 Request

1. GET and POST requests can be created a variety of ways
2. Use AJAX to asynchronously request data from APIs. `fetch()` and `async/await` are new functionalities developed in ES6 (promises) and ES8 respectively
3. Promises are a new type of JavaScript object that represent data that will eventually be returned from a request



4. `fetch()` is a web API that can be used to create requests. `fetch()` will return promises
5. We can chain `.then()` methods to handle promises returned by `fetch()`
6. The `.json()` method converts a returned promise to a JSON object
7. `async` is a keyword that is used to create functions that will return promises
8. `await` is a keyword that is used to tell a program to continue moving through the message queue while a promise resolves
9. `await` can only be used within functions declared with `async`

## 9.5 Errors

Errors will prevent a program from executing unless it is handled

## 9.6 Promise

1. Promises are objects that represent the eventual outcome of an asynchronous operation
2. A Promise object can be in one of three states:
  - Pending
  - Fulfilled
  - Rejected
3. We refer to a promise as settled if it is no longer pending – it is either fulfilled or rejected
4. 

```
const prom = new Promise((resolve, reject) => {
  resolve('Yay!');
});

const handleSuccess = (resolvedValue) => {
  console.log(resolvedValue);
};
```

```
prom.then(handleSuccess); // Prints: 'Yay!'
```

## 5. Using `catch()` with Promises

```
prom
  .then((resolvedValue) => {
    console.log(resolvedValue);
  })
  .then(null, (rejectionReason) => {
    console.log(rejectionReason);
  });
```

Since JavaScript doesn't mind whitespace, we follow a common convention of putting each part of this chain on a new line to make it easier to read. To create even more readable code, we can use a different promise function: `.catch()`.

```
prom
  .then((resolvedValue) => {
    console.log(resolvedValue);
  })
  .catch((rejectionReason) => {
    console.log(rejectionReason);
  });
```

Correct! `.catch(onReject)` is syntactic sugar for `.then(undefined, onReject)`.

## 6. Chaining multiple promises see [javascript.pdf](#) for a demo

## 7. Avoiding Common Mistakes

- Mistake 1: Nesting promises instead of chaining them.
- Mistake 2: Forgetting to **return** a promise.

## 8. Using `Promise.all()`

To maximize efficiency we should use concurrency, multiple asynchronous operations happening together. With promises, we can do this with the function `Promise.all()`

9. A Promise's constructor has a single parameter, called the "executor function".  
The executor function has two parameters – resolve and reject.

## 9.7 Async await

### 9.7.1 The async Keyword

The `async` keyword is used to write functions that handle asynchronous actions. We wrap our asynchronous logic inside a function prepended with the `async` keyword. Then, we invoke that function

```
async function myFunc() {  
  // Function body here  
};
```

```
myFunc();
```

we can also create `async` function expressions

```
const myFunc = async () => {  
  // Function body here  
};
```

```
myFunc();
```

`async` functions always return a promise. This means we can use traditional promise syntax, like `.then()` and `.catch` with our `async` functions. An `async` function will return in one of three ways

- If there's nothing returned from the function, it will return a promise with a resolved value of `undefined`
- If there's a non-promise value returned from the function, it will return a promise resolved to that value

```
async function fivePromise() {  
  return 5;  
}
```

```

fivePromise()
.then(resolvedValue => {
    console.log(resolvedValue);
}) // Prints 5

```

- If a promise is returned from the function, it will simply return that promise

### 9.7.2 The await Operator

`async` functions are almost always used with the additional keyword `await` inside the function body.

The `await` keyword can only be used inside an `async` function. `await` is an operator: it returns the **resolved value** of a promise. Since promises resolve in an indeterminate amount of time, `await` halts, or pauses, the execution of our `async` function until a given promise is resolved.

```

async function asyncPromAll() {
    const resultArray = await Promise.all([asyncTask1(), asyncTask2(),
asyncTask3(), asyncTask4()]);
    for (let i = 0; i<resultArray.length; i++){
        console.log(resultArray[i]);
    }
}

```

## 10 CSS

### 10.1 Caveats

1. The unit `px`, if the value is 0 then you can omit the `px` unit, if it's not 0 then you have to add `px`
- 2.

### 10.2 Media Queries

-

- Rather than set breakpoints based on specific devices (too many different device screen sizes), the best practice is to resize your browser to view where the website naturally breaks based on its content. The dimensions at which the layout breaks or looks odd become your media query breakpoints. Within those breakpoints, we can adjust the CSS to make the page resize and reorganize
- The points at which media queries are set are called breakpoints. For example, if we want to target tablets that are in landscape orientation, we can create the following breakpoint:

```
@media only screen and (min-width: 768px) and (max-width: 1024px) and
(orientation: landscape) {
    /* CSS ruleset */
}
```

and (min-width: 768px) —This part of the rule is called a *media feature*, and instructs the CSS compiler to apply the CSS styles to devices with a width of 768px or larger

### 10.3 CSS Display and Positioning

- 即使position设置为absolute，其宽度用百分比表示时是相对于父节点的宽度而言，见4
- Floated elements must have a width specified
- Float works for static and relative positioned elements
- If you're simply interested in moving an element as far left or as far right as possible on the page, you can use the `float` property

### 10.4 The box model

- All major web browsers have a default stylesheet they use in the absence of an external stylesheet. These default stylesheets are known as *user agent stylesheets*. In this case, the term "user agent" is a technical term for the browser

- The overflow property is set on a parent element to instruct a web browser how to render child elements. For example, if a div's overflow property is set to `scroll`, all children of this div will display overflowing content with a scroll bar.
- Margin collapse: Unlike horizontal margins, vertical margins do not add. Instead, the larger of the two vertical margins sets the distance between adjacent elements (apply to nested elements as well, maybe there's no nested elements at all **I think there is**)

ylx-wx-beta

index.js

```
import React from 'react'
import './index.less'
import SearchHeader from '../.../components/SearchHeader'

export default class TicketSearch extends React.Component {

  search = (e) => {
    console.log(e.target)
  }

  render() {
    return (
      <div className="hot-search-wrap">
        <SearchHeader showCity={false} onChange={this.search} />
        <div className="content-wrap">
          <div className="head">热门搜索</div>
          <div className="locations">
            <button type="button">武汉花世界·花朵萌乐园</button>
            <button type="button">东湖游船</button>
            <button type="button">光谷步行街</button>
            <button type="button">户部巷</button>
            <button type="button">武汉胜天生态农庄</button>
            <button type="button">姚家山风景区</button>
          </div>
        </div>
      </div>
    )
  }
}
```

```
<button type="button">武汉东湖风景区</button>
<button type="button">武汉东湖磨山景区</button>
</div>
```

```
<div className="history">
  <div className="head">历史搜索</div>
</div>
</div>
</div>
)
}
}
```

index.less

```
@import '../.../utils/color.less';
```

```
.hot-search-wrap {
  .content-wrap {
    margin-top: 70px;
    padding: 0 15px;
    .head {
      margin: 15px 0;
    }
    button {
      display: inline-block;
      padding: 6px;
      border-radius: 5px;
      margin-right: 10px;
      margin-bottom: 6px;
      border: 1px solid blue;
    }
  }
}
```

```
}  
}
```

- The `padding` property is often used to expand the background color and make content look less cramped
- By default, the dimensions of an HTML box are set to hold the raw contents of the box

## 10.5 Non-Inherited properties

To name but a few, non-inherited properties are:

`width`, `height`, `padding`, `border`, `margin`, `position`, `background`, etc.

## 10.6 Learn Responsive Design

1. **Historically**, the `em` represented the width of a capital letter M in the typeface and size being used. That is no longer the case. Today, the `em` represents the size of the base font being used. For example, if the base font of a browser is 16 pixels (which is normally the default size of text in a browser), then 1 `em` is equal to 16 pixels. 2 `ems` would equal 32 pixels, and so on
2. A high resolution display may have a `min-resolution` of 150dpi
3. The `height` property is set to `auto`, meaning an image's height will automatically scale proportionally with the width

the last line will display images as block level elements (rather than inline-block, their default state)

```
.container {  
  width: 50%;  
  height: 200px;  
  overflow: hidden;  
}
```

```
.container img {  
  max-width: 100%;
```



```
height: auto;
display: block;
}
```

Images or videos will shrink to the full width of their container, scale proportionally, and display partially if the image dimensions exceed container dimensions

It's worth memorizing the entire example above. It represents a very common design pattern used to scale images and videos proportionally

4. Percentages are often used to size box-model values, like width and height, padding, border, and margins. They can also be used to set positioning properties (top, bottom, left, right). When percentages are used, elements are sized relative to the dimensions of their parent element (also known as a container)

**Note:** Because the box model includes padding, borders, and margins, setting an element's width to 100% may cause content to overflow its parent container. While tempting, 100% should only be used when content will not have padding, border, or margin

5. Rem stands for root em. It acts similar to em, but instead of checking parent elements to size font, it checks the root element. The root element is the `<html>` tag

6. 

```
.splash-section {
    font-size: 18px;
}
```

```
.splash-section h1 {
    font-size: 1.5em;
}
```

Instead, a base font size (18px) is defined for all text within the `splash-section` element. The second CSS rule will set the font size of all `h1` elements inside of `splash-section` relative to the base font of `splash-section` (18 pixels). The resulting font size of `h1` elements will be 27 pixels.

### 10.6.1 Percentages: Padding & Margin

1. When percentages are used to set padding and margin, however, they are calculated based only on the *width* of the parent element.

For example, when a property like `margin-left` is set using a percentage (say 50%), the element will be moved halfway to the right in the parent container (as opposed to the child element receiving a margin half of its parent's margin)

Vertical padding and margin are also calculated based on the width of the parent

## 10.7 display property

**Note:** Setting the display property of an element only changes **how the element is displayed**, NOT what kind of element it is. So, an inline element with `display: block;` is not allowed to have other block elements inside it.

## 10.8 Grid layout

1. flexbox is really meant for only specific use cases, like navigation bars (navbars)

## 10.9 p tag

1. `<figcaption><p>The <em>Cosmos</em> is all there is!</p></figcaption>`  
if you don't add p tag then there will be no space before or after Cosmos

## 11 Node

Enter repl (read evaluate print loop): type node in command line, exit command `process.exit()`

## 12 npm (or yarn)

- 1.
2. If you're using npm 5, you'll probably also see a `package-lock.json` file in your directory

### 3. Useful command `npm show create-react-app version`

Get list of locally installed packages: `npm list`, you can find the version of a specific package by passing its name as an argument. For example, `npm list grunt`

Get list of globally installed packages: `npm list -g --depth 0`

### 4. nvm: node version manager, to manage multiple node versions on one operating system

### 5. To update Node, the most reliable way is to download and install an updated installer package from their website (see link above). To update npm, use the following command in your terminal:

```
npm install npm@latest -g
```

## 13 DOM

### 1. `document.createElement()`

```
<!DOCTYPE html>
<html>
<head>
  <title>||Working with elements||</title>
</head>
<body>
  <div id="div1">The text above has been created dynamically.</div>
</body>
</html>
```

```
document.body.onload = addElement;
```

```
function addElement () {
  // create a new div element
  var newDiv = document.createElement("div");
  // and give it some content
  var newContent = document.createTextNode("Hi there and greetings!");
```

```
// newDiv.textContent = "Hi there and greetings!".

// add the text node to the newly created div
newDiv.appendChild(newContent);

// add the newly created element and its content into the DOM
var currentDiv = document.getElementById("div1");
document.body.insertBefore(newDiv, currentDiv);
}
```

2. **Document** and **window** objects are the objects whose interfaces you generally use most often in DOM programming. In simple terms, the **window** object represents something like the browser, and the **document** object is the root of the document itself. **Element** inherits from the generic **Node** interface, and together these two interfaces provide many of the methods and properties you use on individual elements
3. That is to say, it's *written* in JavaScript, but it uses the DOM to access the document and its elements

## 14 cmd

- 1.
2. [git log output encoding issues on Windows 10 command prompt](#)
3. Windows clear screen commands:

cls (clear screen)    Press Esc key    Ctrl + c

## 15 中文笔记

- 
- 变量提升：JS会在词法分析阶段，把变量的声明语句提升到作用域的顶部

## 16 命名规范 Name convention

- 线路首页: Route or RouteHome
- 
- 

## 17 Abbr

|   |    |                             |     |           |       |
|---|----|-----------------------------|-----|-----------|-------|
| block   | 块级 | inline                      | 内联  | lexical   | 词法作用域 |
| closure   | 闭包 | render                      | 渲染  | container | 容器    |
| content   | 内容 | form                        | 表单  | canvas    | 画布    |
| component   | 组件 | margin                      | 外边距 | style     | 样式    |
| callback  | 回调 | revert                      | 回滚  | collapse  | 折叠    |
| template string 模板字符串                               |    | logical operators 逻辑运算符     |     |           |       |
| object destructure 对象解构                             |    | array destructure 数组解构      |     |           |       |
| logical operands: true false                        |    | SoC: Separation of Concerns |     |           |       |
| POSIX: portable operating system interface for unix |    |                             |     |           |       |
| umd: universal module definition                    |    |                             |     |           |       |
| XSS attack: cross-site scripting attack             |    |                             |     |           |       |