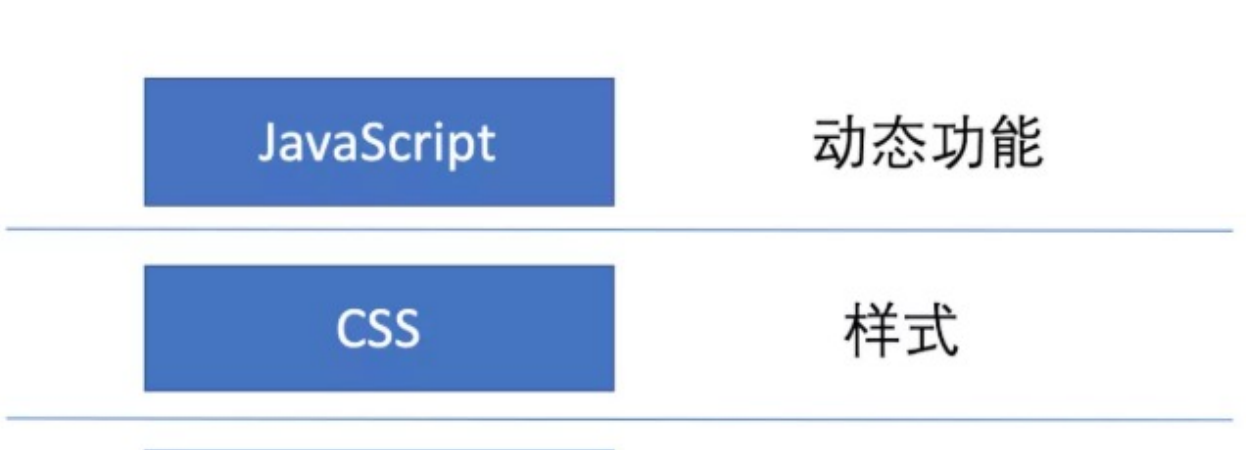
组件实践（3）：组件化样式

在上一节中，我们创造了“秒表”组件，但是那个“秒表”的样式很粗糙，因为我们只实现了功能，而不关注样式。并不是说样式不重要，实际上，样式美观是前端开发的一个重要部分，只是做事有先有后，用渐进式开发的原则，先搞定功能，然后再来处理样式。

在这一节中，我们要探讨如何给 React 组件增加样式，让“秒表”这个应用看起来更美观。

React 带来的对样式管理革命

在我刚入行的时候，业界对前端开发普遍有这样的认知，把网页应用分为三层，分别是用 HTML 实现的“内容”，用 CSS 实现的“样式”，还有用 JavaScript 实现的“动态行为”。



就像上面这样，构建一个网页应用，首先用 HTML 展示内容。比如，我们展示一个新闻，首先用 HTML 的 `p` 标签展示文字，用 `img` 标签展示图片，这样，即使没有 CSS 和 JavaScript，至少用户还可以看见新闻内容。

当然，单纯只有 HTML，那网页内容也太枯燥了，所以还是需要 CSS 来增加一些多彩的样式，修改一下字体、颜色、阴影之类，这是第二层的功能。

最后，进一步提高用户体验，就要 JavaScript 上场了，在 HTML 和 CSS 的基础上赋予网页动态交互的功能，比如给新闻网页增加一个“点赞”的功能，这是 HTML 和 CSS 无法做到的。

按照这样“内容” -> “样式” -> “动态功能”的方式渐进增强（Progressive Enhancement），是非常正确的思想。但是，长期以来，实现方式存在问题，问题就是 HTML、CSS 和 JavaScript 被分开管理了。

如果你在 React 诞生之前从事过网页开发，肯定有这样的体会。为了修改一个功能，需要牵扯到 HTML、CSS 和 JavaScript 的修改，但是这三部分分别属于不同的文件，明明是一个功能，你却要去修改至少三个文件。

在软件开发中，同一个功能相关的代码最好放在一个地方，这就是高内聚性（High Cohesiveness）。把网页功能分在 HTML、CSS 和 JavaScript 中，明显背离了高内聚性的原则，但是我们也忍受了这个做法这么多年，直到 React 出现。

在实现“秒表”的时候，我们已经可以看到，“内容”和“动态功能”已经混合在一起，换句话说，长得很像 HTML 的 JSX 负责产生“内容”，和各种响应用户输入的 JavaScript 代码共同存在于 React 组件之中。

在 React 中，当你要修改一个功能的内容和行为时，在一个文件中就能完成，这样就达到了高内聚的要求。

那么，在 React 中又是如何处理样式的呢?

我们先从组件的 style 属性开始，最后过渡到组件式的样式。

style 属性

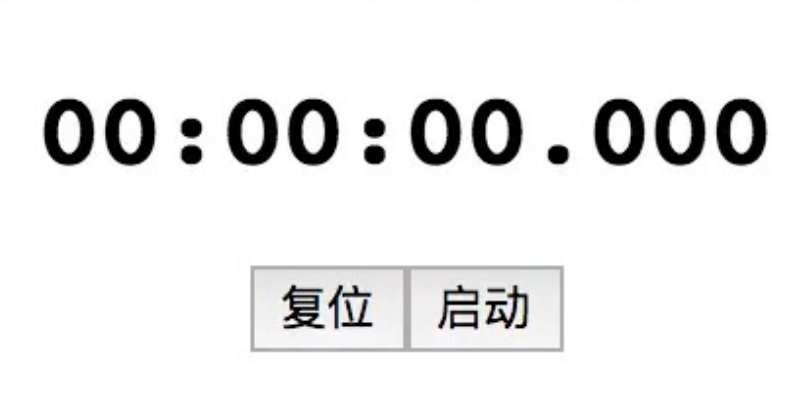
在上一小节实现的“秒表”中，虽然功能齐备，但是展示上有一个大问题，就是当时钟开始运转之后，因为各个数字的宽度不同，比如 `1` 就没有 `0` 宽，导致时间宽度忽大忽小，产生闪烁效果，这样看起来很不专业。

为了解决这个问题，我们就需要定制 MajorClock 的样式。

最简单也是最直接的方法，就是给对应的 React 元素增加 style 属性，属性值为一个普通的 JavaScript 对象，如下所示：

```
const MajorClock = ({milliseconds=0}) => {
  const style = {
    'font-family': 'monospace'
  };
  return <h1 style={style}>{ms2Time(milliseconds)}</h1>
}
```

在上面的例子中，我们把 MajorClock 中的 `h1` 元素的 `font-family` 设为 `monospace`，`monospace` 是等宽字体，这样所有数字所占宽度相同，数字变化起来的时候宽度也就不会发生变化了，效果图如下：



你可能也见过有人像下面这么写：

```
const MajorClock = ({milliseconds=0}) => {
  return <h1 style={{
    'font-family': 'monospace'
  }}>{ms2Time(milliseconds)}</h1>
}
```

上面这种写法，并不好。因为每次渲染 MajorClock 组件都会创建一个新的 style 对象，纯粹就是浪费。

如果 style 对象每次都是一样的，最好把它提取到组件之外，这样就可以重用一个对象，像下面这样：

```
const clockStyle = {
  'font-family': 'monospace'
};

const MajorClock = ({milliseconds=0}) => {
  return <h1 style={clockStyle}>{ms2Time(milliseconds)}</h1>
}
```

导入 CSS 文件

长期来，前端开发者都习惯了使用 CSS 来定制样式，React 也支持这种做法。

我们以 ControlButtons 为例，改进控制按钮的样式。

为了配合 CSS，我们要在 ControlButtons 的 JSX 中让渲染出来的 DOM 元素包含 class。

```
const ControlButtons = ({activated, onStart, onPause, onReset, onSplit}) => {
  if (activated) {
    return (
      <div>
        <button className="left-btn" onClick={onSplit}>计次</button>
        <button className="right-btn" onClick={onPause}>停止</button>
      </div>
    );
  } else {
    return (
      <div>
        <button className="left-btn" onClick={onReset}>复位</button>
        <button className="right-btn" onClick={onStart}>启动</button>
      </div>
    );
  }
};
```

值得一提的是，虽然最终产生的 DOM 或者 HTML 中属性为 class，在 JSX 中不能用 class，要用 className 来指定元素的类名。

然后，我们在 `ControlButtons.js` 中增加下面这样，导入一个同目录下的 `ControlButtons.css` 文件：

```
import "./ControlButtons.css";
```

create-react-app 会用 webpack 完成打包过程，只要 JavaScript 文件中应用的资源，都会被打包进最终的文件，所以，`ControlButtons.css` 中的样式规则就会被应用。

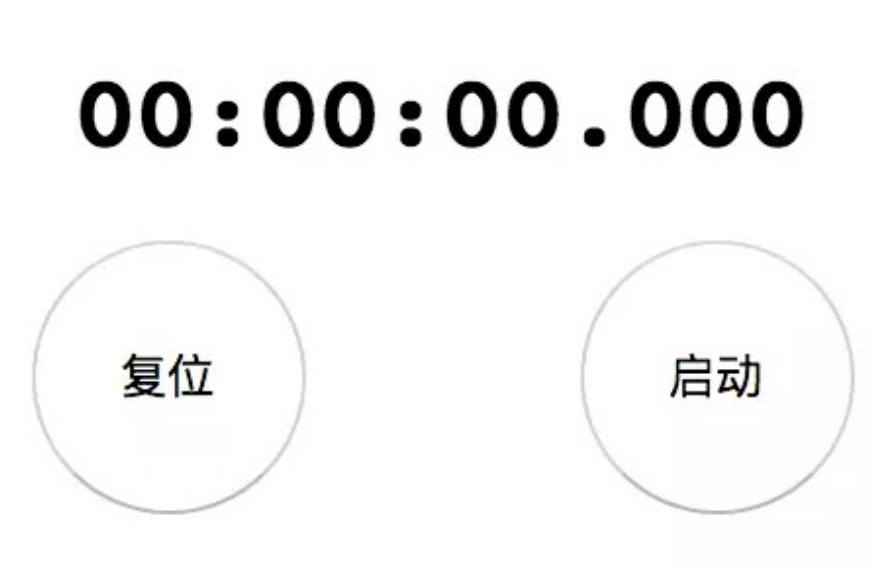
`ControlButtons.css` 中的内容如下：

```
.left-btn, .right-btn {
  border-radius: 50%;
  width: 70px;
  height: 70px;
}

.left-btn {
  margin: 0 35px 0 0;
}

.right-btn {
  margin: 0 0 0 35px;
}
```

最终的效果图如下：



可以看到，按钮之间有了合适的距离，而且边缘也和 iPhone 上的“秒表”一样显示为圆形。

你已经做到接近 iPhone 外观的“秒表”应用了，给自己鼓鼓掌吧！

组件式的样式

对比使用 style 属性和导入 CSS 两种方法，可以看出各有优缺点。

使用 style 属性，好处是可以将样式应用到每个元素，互相不干扰；缺点就是非常不简洁，如果你想要定制一个元素的样式，必须给这个元素加 style 属性。

比如，我们想让 MajorClock 中的 `h1` 元素字体为 `monospace`，使用 style 属性来实现，就要给 `h1` 加上 style，如果只有一个 `h1` 元素还好应付，如果很多 `h1` 元素，就非常麻烦。

```
const style={
  'font-family': 'monospace'
};

  <h1 style={style}>...</h1>
```

相反，用 CSS 表达复杂的样式规则很容易，比如，上一段提到的样式，用 CSS 轻松可以实现，而且不用给每个 `h1` 加什么 style 属性。

```
h1 {
  font-family: monospace;
}
```

不过，CSS 也有它的缺点，CSS 定义的样式是全局的，这样很容易失控，比如上面的 CSS 规则，一旦导入，那么所有的 `h1` 都具备这样的样式，即使不在 MajorClock 中的 `h1` 元素，一样被 MajorClock 导入的 CSS 文件影响。

为了解决不同模块之间 CSS 互相干扰的问题，前端开发者想出了好多种解决方法，基本原则就是给 CSS 规则增加更加特定的限制。比如，要限定上面的 CSS 规则只作用于 MajorClock 中的 `h1` 元素，就要这样来写一个 `MajorClock.css`：

```
.clock h1 {
  font-family: monospace;
}
```

但是，你也需要修改 MajorClock 的 JSX，让 `h1` 包含在一个类名为 `clock` 的元素中。

```
import "./MajorClock.css";

const MajorClock = ({milliseconds=0}) => {
  return (
    <div class="clock">
     <h1 style={clockStyle}>{ms2Time(milliseconds)}</h1>
    </div>
  );
}
```

这样当然可行，但是，开发者不好处理 JSX 和 CSS 之间的关系，而且，就像在上面说过的，这样违背高内聚的原则。当你需要修改一个组件时，要被迫去分别修改 JavaScript 文件 和 CSS 文件，明显不是最优的方法。

我曾说过，在 React 的世界中，一切都是组件，所以很自然诞生了组件化的样式(Component Style）。

组件化样式的实现方式很多，这里我们介绍最容易理解的一个库，叫做 `styled-jsx`。

添加 styled-jsx 支持

要使用 styled-jsx，必须要修改 webpack 配置，一般来说，对于用 create-react-app 创建的应用，需要用 `eject` 方法来“弹射”出配置文件，只是，`eject` 指令是不可逆的，不到万不得已，我们还是不要轻易“弹射”。

一个更简单的方式，是使用 `react-app-rewired`，不需要 `eject`，轻轻松松就能够修改 create-react-app 产生应用的配置方法。

首先，我们在项目中安装 react-app-rewired 和 styled-jsx。如果读者使用的是npm v5之前的版本，最好添加 `--save` 参数用于修改 `package.json`，如果使用npm v5之后版本，则无需添加 `--save` 参数。

```
npm install react-app-rewired styled-jsx
```

然后，打开 `package.json` 文件，找到 `scripts` 这个部分，应该是下面这样：

```
   "scripts": {
     "start": "react-scripts start",
     "build": "react-scripts build",
     "test": "react-scripts test --env=jsdom",
     "eject": "react-scripts eject"
   }
```

当我们在命令行执行 `npm start` 时，执行的就是 `scripts` 部分定义的指令，可以看到都是执行 `react-scripts`。

在这里还可以看到 `eject` 指令的定义，我们做这个修改，恰恰就是为避免使用 `eject`。

我们修改 `scripts` 部分的代码如下：

```
   "scripts": {
    "start": "react-app-rewired start",
    "build": "react-app-rewired build",
    "test": "react-app-rewired test --env=jsdom",
    "eject": "react-scripts eject"
```

修改的方法其实就是把 `start`、`build` 和 `test` 对应脚本中的 `react-scripts` 替换为 `react-app-rewired`，之后，当用 npm 执行这些指令的时候，就会使用 react-app-rewired。

react-app-rewired 扩展了 react-scripts 的功能，可以从当前目录的 `config-overrides.js` 文件中读取配置，扩充 react-scripts 的功能。

我们需要让 react-scripts 支持 styled-jsx，对应只需要在项目根目录增加一个 `config-overrides.js` 文件，内容如下：

```
const { injectBabelPlugin } = require('react-app-rewired');

module.exports = function override(config, env) {
  config = injectBabelPlugin(['styled-jsx/babel'], config);

  return config;
};
```

上面 config-overrides.js 文件中的逻辑很简单，就是把 `styled-jsx/babel` 注入到 react-scripts 的基本配置中去，然后，我们的应用就支持 styled-jsx 了。

使用 styled-jsx 定制样式

有了 styled-jsx 中，我们就可以在 JSX 中用 `style jsx` 标签直接添加 CSS 规则。

比如，我们要给 MajorClock 中的 `h1` 增加 CSS 规则，可以这样使用：

```
const MajorClock = ({milliseconds=0}) => {
  return (
    <React.Fragment>
      <style jsx>{`
        h1 {
          font-family: monospace;
        }
      `}</style>
      <h1>
        {ms2Time(milliseconds)}
      </h1>
    </React.Fragment>
  );
};
```

注意紧贴 style jsx 内部的是一对大括号，大括号代表里面是一段 JavaScript 的表达式，再往里，是一对符号，代表中间是一段多行的字符串，也就是说，style jsx 包裹的是一个字符串表达式，而这个字符串就是 CSS 规则。

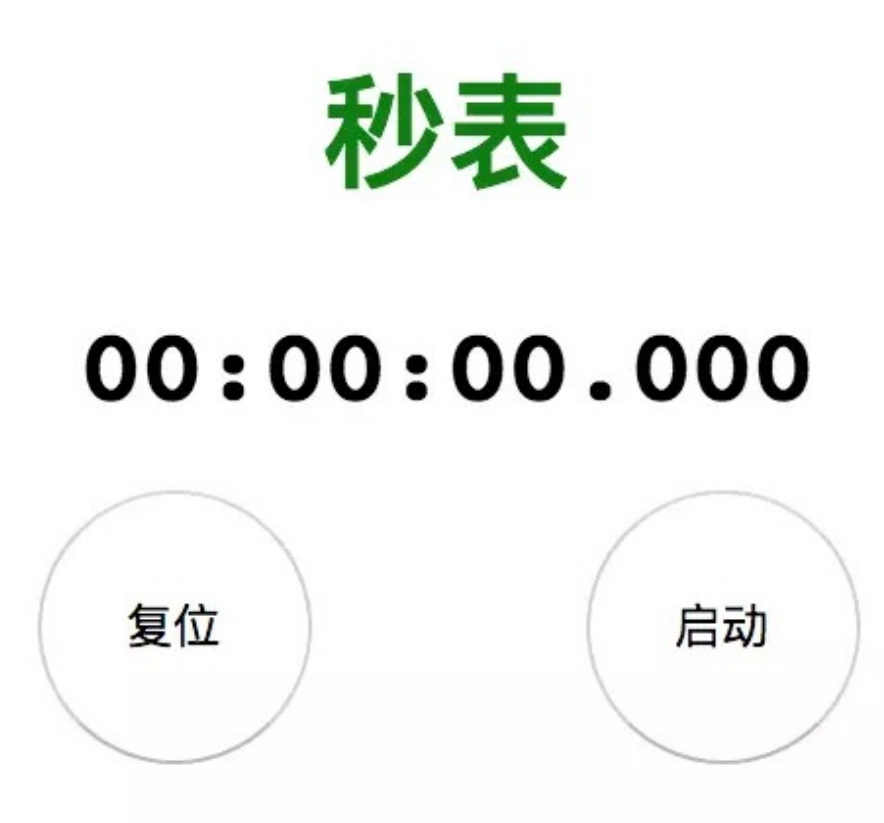
在 MajorClock 中用 style jsx 添加的 CSS 规则，只作用于 MajorClock 的 JSX 中出现的元素，不会影响其他的组件。

你可以尝试在其他组件中添加 `h1` 元素，也可以尝试在其他组件中添加 `style jsx` 标签来定制 `h1` 的样式，会发现和 MajorClock 完全是井水不犯河水，互不影响。

我在 StopWatch 中添加一个 `h1` 元素，内容就是“秒表”，然后用 `style jsx` 把 `h1` 的颜色设为绿色，代码如下：

```
  render() {
    return (
      <Fragment>
        <style jsx>{`
          h1 {
            color: green;
          }
        `}</style>
        <h1>秒表</h1>
        <MajorClock
          milliseconds={this.state.currentTime - this.state.startTime}
          activated={this.state.isStarted}
        />
   ...
```

界面效果如下，可以看到，StopWatch 中的 `h1` 字体不是 monospace，MajorClock 中的 color 也不是绿色。



可见，styled jsx 中虽然使用了 CSS，但是这些 CSS 规则只作用于所在组件中的样式，甚至不会影响子组件的样式。

这样一来，我们既可以使用 CSS 的语法，又可以把 CSS 的作用域限定在一个组件之内，达到了高内聚的要求。

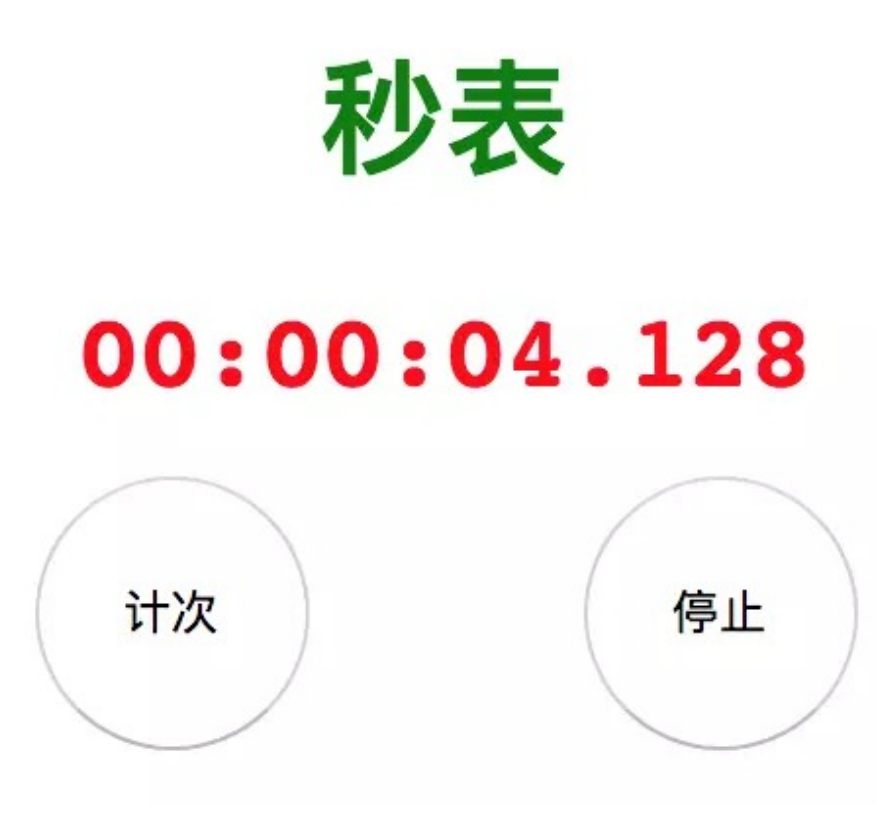
动态 styled jsx

更妙的是，我们还可以动态修改 styled jsx 中的值，因为 styled jsx 的内容就是字符串，我们只要修改其中的字符串，就修改了样式效果。

比如，我们让 MajorClock 在开始计时状态显示红色，否则显示黑色，修改代码如下：

```
const MajorClock = ({milliseconds=0, activated=false}) => {
  return (
    <React.Fragment>
      <style jsx>{`
        h1 {
          color: ${activated? 'red' : 'black'};
          font-family: monospace;
        }
      `}</style>
      <h1>
        {ms2Time(milliseconds)}
      </h1>
    </React.Fragment>
  );
};
```

在 style jsx 中，`color` 后面的值不是固定的，利用 ES6 的字符串模板功能，我们可以根据 activated 的值动态决定是 red 还是 black，最终效果图如下：



大家可以尝试用 style jsx 进一步定制“秒表”的样式，从中体会“组件式样式”的方便之处。

小结

本小节介绍了 React 中的样式实现方法，读者应该掌握：

1. React 将内容、样式和动态功能聚集在一个模块中，是高聚合的表现；
2. React 原生 style 属性的用法；
3. 组件化样式 styled jsx 的用法。

留言

评论将在后台进行审核，审核通过后对所有人可见

zhangsai0218
style jsx 中的CSS规则，在编辑器里有高亮插件吗?
▲ 0 评论 3天前

ggstudy
react 有没有比较优雅的方式将css文件提出来，加上scoped，不与组件混合在一个组件内，因为有的时候一些组件业务代码偏多，没有必要提取成小组件，但是css写在组件中导致组件的代码很长，不好维护，类似vue
▲ 0 评论 12天前

dpyzo0o
react-app-rewired 不再支持CRA 2.0了，可以考虑用craco
▲ 0 评论 14天前

AboyL 前端开发者 @ 保密
把样式都写在style里面，跟把样式写在css里面，从性能上来说会下降吗？如果不考虑文件大小得话？
▲ 0 收起评论 20天前

程墨 Hulu
相关性能的影响基本上就是九牛一毛，而且，改变行为（js）和展示（HTML）
但是不需要改变样式（CSS）的机会比较小，既然要改都要重新下载CSS，也就不存在充分利用cache带来的性能好处了。
19天前

评论审核通过后显示 评论

仙道CM 前端
个人感觉就是配置的时候不是那么优雅，可读性有待商榷，但是从高内聚角度来说比style-components做的更彻底，也更优雅
▲ 0 评论 26天前

老腊肉学长 全栈工程师 @ Tencent
我猜“高内聚低耦合”的样式底层是使用带有scoped属性的<style>标签。vue文件原生支持。
▲ 0 评论 27天前

zb本尊
我想知道为什么用了组件化样式，但css规则却不仅仅只用于当前组件，扩散到了全局
▲ 0 评论 27天前

yadong 前端开发 @ 知乎
“和各种响应用户输入的 JavaScript 代码共同存在于 Reac 组件之中”，这段文字React拼错了
▲ 0 评论 29天前

得德 web developer
感觉这种写法，可读性是差了好多，并能一眼看懂dom结构了
▲ 0 评论 1月前

Tao在掘金
这种写法未免也太过繁琐了吧 为什么不使用 styled-component来实现呢
▲ 2 评论 1月前

东东东寅
其实有点像vue的.vue文件结构吧
▲ 0 评论 1月前

Jexxie
这个秒表停止再启动，是重新计时的，请问该如何实现保存计时状态？
▲ 0 收起评论 1月前

十二不死灰 前端 @ 上海汉得研发
pauseTemp：0；暂停时用个变量去存。pauseTemp: currentTime - startTime + pauseTemp. render里 const milliseconds = isStarted ? currentTime - startTime + pauseTemp : pauseTemp;
1月前

评论审核通过后显示 评论

静水流深邃 web前端工程师 @ come on
感觉styled jsx 会让代码可读性变差，为什么不同less或者sass，同问？
▲ 0 收起评论 1月前

程墨 Hulu
如果要用less或者sass完全可以啊，至于style不同方案的可读性的问题，业界也有很大争议，按照最习惯的方式写就好了，也不是说styled jsx就适合所有人。
1月前

评论审核通过后显示 评论

大妮爸
讲的蛮好，由浅入深，由表及里
▲ 0 收起评论 1月前

程墨 Hulu
谢谢支持
1月前

评论审核通过后显示 评论

Vinci
大佬求源码
▲ 0 收起评论 1月前

程墨 Hulu
放到这里了 https://github.com/mocheng/react-practice
1月前

评论审核通过后显示 评论

undefined1473484942981
花里胡哨，dom复杂了怎么搞
▲ 0 收起评论 1月前

程墨 Hulu
抱歉，我没有明白你的观点。
1月前

XYSC
styled jsx没有style-components好用，也不够简洁明了
1月前

程墨 Hulu
回复 XYSC：嗯，每个工具都有优缺点，萝卜白菜各有所爱
1月前

评论审核通过后显示 评论

Ru_minnnn
高级
▲ 0 评论 1月前

孤落无痕 前端开发
为什么不使用less/sass
▲ 1 收起评论 1月前

程墨 Hulu
完全可以，对于习惯sass或者less的开发者，可以继续使用这样的工具，但是如我在小册中所说，React的哲学就是把内容、样式和行为聚合在一个模块里，用组件化样式的方法更加易于管理。
1月前

乔同志
我也是比较习惯用scss
1月前

评论审核通过后显示 评论