



王下邀月熊 Chevalier 21.8k

如何写出漂亮的React组件

react.js 发布于 2016-11-21 • 约 17 分钟

本文翻译自[Make-Your-React-Components-Pretty](#)。欢迎转载，注明出处。

本文从属于笔者的[Web前端入门与最佳实践](#) 中的[React入门与最佳实践](#)系列，同类型文章还包括[React 代码风格约定](#)。

```
19
20  const SecondaryResults = ({  
21    secondaryItems,  
22    secondaryItemCount,  
23    query,  
24    strategyId,  
25    onSeeSimilarClick,  
26    children  
27  }) => (  
28    <div className="search-result-listview-secondary-items">  
29      <div className="search-result-divider"></div>  
30      <div className="secondary-search-result-description">  
31        <div className="secondary-search-result-count">  
32          Similar to "{query}"  
33        </div>  
34        <div className="secondary-search-result-count-text">  
35          {secondaryItems.length} of {secondaryItemCount} items  
36        </div>  
37      </div>  
38    </div>
```

在[Walmart Labs](#)的产品开发中，我们进行了大量的Code Review工作，这也保证了我有机会从很多优秀的工程师的代码中学习他们的代码风格与样式。在这篇博文里我会分享出我最欣赏的五种组件模式与代码片。不过我首先还是要谈谈为什么我们需要执着于提高代码的阅读体验。就好像你有很多种方式去装扮一只猫，如果你把你的爱猫装扮成了如下这样子：



你或许可以认为萝卜青菜各有所爱，但是代码本身是应当保证其可读性，特别是在一个团队中，你的代码是注定要被其他人阅读的。电脑是不会在意这些的，不管你朝它们扔过去什么，它们都会老老实实的解释，但是你的队友们可不会这样，他们会把丑陋的代码扔回到你的脸上。而所谓的Pretty Components，应该包含如下的特性：

- 即使没有任何注释的情况下也易于理解
- 比乱麻般的代码有更好的性能表现
- 更易于进行Bug追溯
- 简洁明了，一句顶一万句

“Less is more.”

— Mies van der Rohe

SFC: Stateless Functional Component

我觉得我们在开发中经常忽略掉的一个模式就是所谓的Stateless Functional Component，不过这是我个人最爱的React组件优化模式，没有之一。我喜爱这种模式不仅仅因为它们能够减少大量的模板代码，而且因为它们能够有效地提高组件的性能表现。总而言之，SFC能够让你的应用跑的更快，长的更帅。



直观来看，SFC就是指那些仅有一个渲染函数的组件，不过这简单的改变就可以避免很多的无意义的检测与内存分配。下面我们来看一个实践的例子来看下SFC的具体作用，譬如：

Related Searches

[smart tv](#)
[samsung tv](#)

[tv stand](#)
[tv wall mount](#)

[vizio tv](#)
[lg tv](#)

[refurbished tv](#)
[apple tv](#)

如果我们用正统的React组件的写法，可以得出如下代码：

```
constructor(props) {
  super(props);
  this._handleClick = this._handleClick.bind(this);
}
_handleClick(suggestedUrl, event) {
  event.preventDefault();
  this.props.onClick(suggestedUrl);
}
render() {
  return (
    <section className="related-search-container">
      <h1 className="related-search-title">Related Searches:</h1>
      <Layout x-small={2} small={3} medium={4} padded={true}>
        {this.props.relatedQueries.map((query, index) =>
          <Link
            className="related-search-link"
            onClick={(event) =>
              this._handleClick(query.searchQuery, event)}
            key={index}>
            {query.searchText}
          </Link>
        )}
      </Layout>
    </section>
  )
}
```

而使用SFC模式的话，大概可以省下29%的代码：

```

const _handleClick(suggestedUrl, onClick, event) => {
  event.preventDefault();
  onClick(suggestedUrl);
};

const RelatedSearch = ({ relatedQueries, onClick }) =>
  <section className="related-search-container">
    <h1 className="related-search-title">Related Searches:</h1>
    <Layout x-small={2} small={3} medium={4} padded={true}>
      {relatedQueries.map((query, index) =>
        <Link
          className="related-search-link"
          onClick={(event) =>
            _handleClick(query.searchQuery, onClick, event)}
          key={index}>
          {query.searchText}
        </Link>
      )}
    </Layout>
  </section>
export default RelatedSearch;

```

代码量的减少主要来源两个方面：

- 没有构造函数（5行）
- 以Arrow Function的方式替代Render语句（4行）

实际上，SFC最迷人的地方不仅仅是其代码量的减少，还有就是对于可读性的提高。SFC模式本身就是所谓纯组件的一种最佳实践范式，而移除了构造函数并且将_handleClick()这个点击事件回调函数提取出组件外，可以使JSX代码变得更加纯粹。另一个不错的地方就是SFC以Arrow Function的方式来定义了输入的Props变量，即以Object Destructuring语法来声明组件所依赖的Props：

```
const RelatedSearch = ({ relatedQueries, onClick }) =>
```

这样不仅能够使组件的Props更加清晰明确，还能够避免冗余的this.props表达式，从而使代码的可读性更好。



最后，我还想要强调下虽然我很推崇SFC，不过也不能滥用它。最合适使用SFC的地方就是之前你用纯组件的地方。在Walmart Labs中，我们使用Redux来管理应用的状态，也就意味着我们绝大部分的组件都是纯组件，也就给了SFC广阔的应用空间。一般来说，有以下特征的组件式绝对不适合使用SFC的：

- 需要自定义整个组件的生命周期管理
- 需要使用到refs

Conditional Components

JSX本身不支持if表达式，不过我们可以使用逻辑表达式的方式来避免将代码切分到不同的子模块中，大概是如下样子：

```
render() {
  <div class="search-results-container">
    {this.props.isGrid
      ? <SearchResultsGrid />
      : <SearchResultsList />}
  </div>
}
```

这种表达式在二选一渲染的时候很有效果，不过对于选择性渲染一个的情况很不友好，譬如如下的情况：

```
render() {
  <div class="search-results-list">
    {this.props.isSoftSort
      ? <SoftSortBanner />
      : null}
  </div>
}
```

这样子确实能起作用，不过看上去感觉怪怪的。我们可以选用另一种更加语义化与友好的方式来实现这个功能，即使用逻辑与表达式然后返回组件：

```
render() {
  <div class="search-results-list">
    {!!this.props.isSoftSort && <SoftSortBanner />}
  </div>
}
```

不过这一点也是见仁见智，每个人按照自己的喜好来就行了。

Arrow Syntax In React And Redux

ES2015里包含了不少可口的语法糖，我最爱的就是那个[Arrow Notation](#)。这个特性在编写组件时很有作用：

```
const SoftSort = ({ hardSortUrl, sortByName, onClick }) => {
  return (
    <div className="SearchInfoMessage">
      Showing results sorted by both Relevance and {sortByName}.
      <Link
        href={`#${hardSortUrl}`}
        onClick={(ev) => onClick(ev, hardSortUrl)}>
        Sort results by {sortByName} only
      </Link>
    </div>
  );
};
```

该函数的功能就是返回JSX对象，我们也可以忽略return语句：

```
const SoftSort = ({ hardSortUrl, sortByName, onClick }) =>
  <div className="SearchInfoMessage">
    Showing results sorted by both Relevance and {sortByName}.
    <Link
      href={`#${hardSortUrl}`}
      onClick={(ev) => onClick(ev, hardSortUrl)}>
      Sort results by {sortByName} only
    </Link>
  </div>
```

代码行数又少了不少咯！