

# 15 | 不可变数据：为什么对React这么重要？

2022-10-04 宋一玮 来自北京



天下无鱼

<https://shikey.com/>

《现代React Web开发实战》

[课程介绍 >](#)



讲述：宋一玮

时长 13:00 大小 11.87M



你好，我是宋一玮，欢迎回到 React 应用开发的学习。

上节课我们不再依赖 CRA，从零开始用 Vite 搭建了一个新的 React 项目，并把 oh-my-kanban 的代码迁移了过来，熟悉了与 React 应用代码直接相关的工程化概念和工具。其中，我们也重点介绍了代码静态检查工具的用法和部分规则。结合从第 3 节课以来学到的知识，到现在你已经基本可以独立开发小型 React 项目了。

从这节课开始，我们将进入新的模块，学习一些大中型 React 项目中会用到的技术和最佳实践，尤其会重点讲解当你融入一个前端开发团队时，需要的开发工作思路和方式的转变，这会帮你更从容应对团队协作。

这节课的主要内容是不可变数据。

没能以正确方式变更数据，是 React 开发中产生 Bug 的重要原因之一。请你回忆一下在 [第 3 节课](#) 末尾，在更新 `todoList` state 时留下的伏笔：`setTodoList(currentTodoList => [newCard, ...currentTodoList])` 为什么不能写成 `todoList.unshift(newCard)` 呢？当学习了不可变数据的原理和实现，你将对 React 的渲染与数据之间的关系更有把握。

下面开始这节课的内容。

## 什么是不可变数据？

**不可变数据（Immutable Data）** 在创建以后，就不可以再被改变。这种数据在编程和调试时更容易预测，有利于降低复杂性。同时在 Web 领域，类似监听数据变化这样的功能非常有用，但运行起来可能会比较重，而不可变数据可以简化实现，降低成本。

目前为止我用过的最方便的不可变数据是基于 JVM 的 Groovy 语言，以下是一段来自 Groovy 官方文档的 [例子](#)（有修改）：

 复制代码

```
1 @groovy.transform.Immutable(copyWith = true)
2 class Customer {
3     String name
4     int age
5     Date since
6     Collection favItems
7 }
8 def d = new Date()
9 def c1 = new Customer(name: '张三', age: 21, since: d, favItems: ['读书', '电影'])
10
11 c1.age = 25 // 抛错
12 def c1Mutated = c1.copyWith(age: 25)
13 assert c1 != c1Mutated // true
14 assert c1.age == 21 // true
15 assert c1Mutated.age == 25 // true
16 c1.favItems << '烫头' // 抛错
17
18 def c2 = new Customer('张三', 21, d, ['读书', '电影'])
19 assert c1 == c2 // true
20 assert c1 != c2 // true
```

当然，我们这节课并不需要你对 Groovy 语言有多少了解，这段代码放在这里，只是为了能用一目了然的方式给你展示不可变数据的特征：

- 不可变数据对象只能在创建时为属性赋值，创建后就不能修改；
- 不可变数据对象的属性也应该是不可变数据，即整个对象树都不可变；
- 变更（**Mutate**）不可变数据只能通过创建新对象、同时显式地指定需要变更的属性的方式，创建出的新对象依旧不可变；
- 用相同属性创建出来的两个同类型的不可变对象，它们逻辑上相等但对象引用是不相等的。

这里我们稍微做一些扩展，回来看 JavaScript，JS 里有没有不可变数据？有。所有原始数据类型（**Primitive Types**）都是不可变数据类型，包括：undefined、null、boolean、number、BigInt、string、Symbol。但对引用类型，如 Object、Function、Array、Map、Set、Date 等，就不是不可变类型了。

即便 JavaScript 没有 Groovy 那么方便的装饰器，我们还是值得在 JS 里探索不可变数据。这主要考虑到它带来的好处：

- 编写纯函数（**Pure Function**）更容易；
- 可以避免函数对传入参数的一些副作用；
- 检测数据变化更轻量更快；
- 缓存不可变数据更安全；
- 保存一份数据的多个版本变得可行。

## React 为什么需要不可变数据？

我们的专栏重点依旧在 React 应用开发上。为什么说 React 需要不可变数据？

这主要还是因为 React 是声明式的框架，为了更新用户看到的页面，我们需要让开发出来的 React 组件响应数据流的变化。这就是说无论开发者，还是 React 框架本身都关注 props、state、context 的数据是否有变化。而前面也讲到了，**对 React 框架，不可变数据可以简化比对数据的实现，降低成本；对开发者，不可变数据在开发和调试过程中更容易被预测。**

接下来看一下 React 在哪些环节会检查数据的变化。

## 协调过程中的数据对比

首先是最核心的 **Fiber** 协调引擎，常提到的 **Diffing** 对比算法就在引擎里，这些对比绝大部分都是在渲染阶段发生的。

我们曾在 [🔗第 12 节课](#) 提到过，**React** 是用 `Object.is()` 方法来判断两个值是否相等的。在以下过程中，**React** 会调用 `is(oldValue, newValue)` 来对比新旧值：

- 更新 **state** 时，只有新旧 **state** 值不相等，才把 **Fiber** 标记为收到更新；
- 更新 **Context.Provider** 中的 **value** 值；
- 检查 **useEffect**、**useMemo**、**useCallback** 的依赖值数组，只有每个值的新老值都检查过，其中有不同时，才执行它们的回调；
- **useSyncExternalStore** 中检查来自外部的应用状态（比如 **Redux**）是否有变化，才把 **Fiber** 标记为收到更新。

还有一种情况是对新旧两个对象做浅对比（**Shallow Compare**），具体实现方式依然是基于 `Object.is()`。当两个对象属性数量相同，且其中一个对象的每个属性都与另一个对象的同名属性相等时，这两个对象才算相等。在下面的过程中，**React** 会调用 `shallowEqual(oldObj, newObj)` 来对比新旧对象（主要是 **props**）：

- **React.memo** 进入更新阶段，如果属性均相同，则跳过该组件继续执行下一个工作；
- **PureComponent** 进入更新阶段，如果属性均相同，则跳过该组件继续执行下一个工作。

以上这两个过程都属于纯组件，我们马上会学习到。

## 合成事件中的数据对比

除了协调引擎，还有一处数据对比发生在合成事件中：在触发 **onSelect** 合成事件前，**React** 用浅对比判断选中项是否真的有变化，真有变化才会触发事件，否则不会触发。

如果你还知道其他数据对比的地方，欢迎在留言区讨论。

## React 纯组件

前面的 [🔗第 9 节课](#) 介绍纯函数与 **React** 组件的关系时，新造了一个名词“纯函数组件”。当时提到：

“纯函数组件”不等同于“纯组件”。因为在 React 里，**纯组件 PureComponent** 是一个主要用于性能优化的独立 API：当组件的 **props** 和 **state** 没有变化时，将跳过这次渲染，直接沿用上次渲染的结果。

到这里，我们终于要介绍纯组件了。首先要注意，纯组件只应该作为**性能优化**的手段，开发者不应该将任何业务逻辑建立在到纯组件的行为上。有两个 API 可以创建纯组件，这里只介绍适合函数组件使用的 `React.memo`，至于 `React.PureComponent`，它是类组件专用，你可以参考 [官方文档](#)。

## React.memo

 复制代码

```
1  const MyPureComponent = React.memo(MyComponent);
2  //      -----
3  //      ^
4  //      |
5  //      纯组件      组件
6
7  const MyPureComponent = React.memo(MyComponent, compare);
8  //      -----
9  //      ^      ^      ^
10 //      |      |      |
11 //      纯组件  组件  自定义对比函数
```

这个 API 第一个参数是一个组件，函数组件或类组件都可以。它会返回一个作为高阶组件的纯组件，这个纯组件接受的 **props** 与原组件相同。每次渲染时纯组件会把 **props** 记录下来，下次渲染时会用新的 **props** 与老的 **props** 做浅对比，如果判断相等则跳过这次原组件的渲染。

但要注意，原组件内部不应该有 **state** 和 **context** 操作，否则就算 **props** 没变，原组件还是有可能因为 **props** 之外的原因重新渲染。

当你不满足于浅对比时，你还可以给这个 API 传入第二个可选参数，一个 **compare** 函数，**compare** 函数被调用时会接受 **oldProps** 和 **newProps** 两个参数，如果返回 **true**，则视为相等，反之则视为不等。

## 不可变数据的实现

刚才提到 JS 中的引用类型并不是不可变的。那如果想用它们，该怎么为它们加入不可变特性呢？

## 手工实现

其实你在前面 oh-my-kanban 项目中已经有经验了，这里再罗列部分：

 复制代码

```
1 // 数组
2 const itemAdded = [...oldArray, newItem];
3 const itemRemoved = oldArray.filter(item => item !== newItem);
4
5 // 对象
6 const propertyUpdated = { ...oldObj, property1: 'newValue' };
7
8 // Map
9 const keyUpdated = new Map(oldMap).set('key1', 'newValue');
```

要领就是“**别.改.原.对.象**”。

## 借助 Helper 库

上面的手工实现在处理复杂对象时，很容易写错，有 [🔗 第三方库](#) 对此做了抽象，但目前基本已经不再维护了。

 复制代码

```
1 import update from 'immutability-helper';
2
3 const state1 = ['x'];
4 const state2 = update(state1, {$push: ['y']}); // ['x', 'y']
```

## 可持久化数据结构和 Immutable.js

到这里，我们就不得不提一个概念：可持久化数据结构（Persistent data structure），它可谓是不可变对象的挚友亲朋。

在计算机编程中，可持久化数据结构（Persistent data structure）是一种能够在修改之后其保留历史版本（即可以在保留原来数据的基础上进行修改——比如增添、删除、赋值）的数

据结构。这种数据结构实际上是不可变对象，因为相关操作不会直接修改被保存的数据，而是会在原版本上产生一个新分支。

——维基百科（[🔗可持久化数据结构](#)）

在 JS 中，可持久化数据结构的代表性实现，就是 FB 开源的 [🔗immutable.js](#)。这个库提供了 List、Stack、Map、OrderedMap、Set、OrderedSet 和 Record 这些不可变数据类型。用这些类型 API 创建的数据，就是基于可持久化数据结构的不可变数据，可以直接用在 React 中。

这里贴两段官方样例代码。首先是神似 JS Array 的 List，你可以看到对 List 对象每个操作都会创建新的 List：

 复制代码

```
1 const { List } = require('immutable');
2 const list1 = List([1, 2]);
3 const list2 = list1.push(3, 4, 5);
4 const list3 = list2.unshift(0);
5 const list4 = list1.concat(list2, list3);
6 assert.equal(list1.size, 2);
7 assert.equal(list2.size, 5);
8 assert.equal(list3.size, 6);
9 assert.equal(list4.size, 13);
10 assert.equal(list4.get(0), 1);
```

还有这个库的强项嵌套结构，在对象树深处的更新也会返回新的不可变对象：

 复制代码

```
1 const { fromJS } = require('immutable');
2 const nested = fromJS({ a: { b: { c: [3, 4, 5] } } });
3
4 const nested2 = nested.mergeDeep({ a: { b: { d: 6 } } });
5 // Map { a: Map { b: Map { c: List [ 3, 4, 5 ], d: 6 } } }
6
7 console.log(nested2.getIn(['a', 'b', 'd'])); // 6
8
9 const nested3 = nested2.updateIn(['a', 'b', 'd'], value => value + 1);
10 console.log(nested3);
11 // Map { a: Map { b: Map { c: List [ 3, 4, 5 ], d: 7 } } }
12
13 const nested4 = nested3.updateIn(['a', 'b', 'c'], list => list.push(6));
14 // Map { a: Map { b: Map { c: List [ 3, 4, 5, 6 ], d: 7 } } }
```



## 如何解决原理和直觉的矛盾？

Immutable.js 很强大，在 React 技术社区也受到过追捧。然而，不知道你平时是怎么使用的，我反正在 React 项目中使用这个框架时，总是要时刻提醒自己，什么时候可以使用 JS 原生的数据类型，什么时候就必须切换到不可变数据类型，这增加了我在开发过程中的认知负荷。

在认知心理学中，认知负荷（Cognitive Load）是指工作记忆资源的使用量。

这虽然会提高程序运行效率，但同时也会降低开发者的开发效率。

那么有没有一种方式，既可以沿用熟悉的 JS 数据类型和方法，又可以类似这节课开头 Groovy 那样，优雅地加入不可变性？

有的，Immer（[🔗 官网](#)）就是这样一款框架，它可以让 JS 开发者使用原生的 JS 数据结构，和本来不具有不可变性的 JS API，创建和操作不可变数据。

以下是来自 Immer 官网的一段样例代码，它的 produce API 接受原数据和数据变更回调函数两个参数，在回调函数中发生的变更，并不会修改原数据本身，而是会返回一个等同于变更结果的新数据：

 复制代码

```
1 import produce from "immer"
2
3 const nextState = produce(baseState, draft => {
4   draft[1].done = true
5   draft.push({title: "Tweet about it"})
6 })
```

## 在 React 中使用 Immer

在函数组件中，可以直接使用 Immer 提供的 Hooks 来替代 useState。

安装 Immer：

 复制代码

```
1 npm install immer use-immmer
```



在组件中使用 Immer:

 复制代码

```
1 import React from "react";
2 import { useImmer } from "use-immer";
3
4 function App() {
5   const [showAdd, setShowAdd] = useState(false);
6   const [todoList, setTodoList] = useImmer([
7     { title: '开发任务-1', status: '22-05-22 18:15' },
8     { title: '开发任务-3', status: '22-05-22 18:15' },
9     { title: '开发任务-5', status: '22-05-22 18:15' },
10    { title: '测试任务-3', status: '22-05-22 18:15' }
11  ]);
12  // ...
13  const handleSubmit = (title) => {
14    setTodoList(draft => {
15      draft.unshift({ title, status: new Date().toLocaleDateString() });
16    });
17  };
18  // ...
```

在后面的课程中，我们还会有一些样例代码中会用到 Immer，届时会介绍更多实用技巧。

## 小结

这节课我们学习了不可变数据，了解了不可变数据在创建后就不能被改变，这样的特性有助于提升 React 对比新旧 props、state、context 数据的效率，而且更容易被预测，有助于开发调试。

然后我们学习了用 React.memo 创建具有更佳性能的纯组件，有关纯组件我也给你分享了一些注意点，要是你记不太清了，可以回去巩固一下。最后介绍了在 JS 中实现不可变数据的几种方式，除了我们在 oh-my-kanban 中的手工实现，还有 Immutable.js 和 Immer 这些开源框架。

接下来我们会用两节课的时间，介绍 React 的应用状态管理，学习什么情况下使用 React 的 state，什么情况下使用外部的应用状态管理框架。


## 思考题

1. 其实在 JavaScript 里有一个方法 `Object.freeze()`，它可以用于不可变数据吗？

2. 在 **React** 相关的技术社区也有不少关于 **ES6 Proxy** 的讨论，你了解或使用过 **Proxy** 吗？如果有过的话，请你设想一下，如果在 **React** 项目中使用 **Proxy**，你会怎样使用它？

好的，这节课内容就到这里。我们下节课再见。

分享给需要的人，Ta购买本课程，你将得 **18** 元

 生成海报并分享

 赞 0  提建议

© 版权归极客邦科技所有，未经许可不得传播售卖。页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

上一篇 直播加餐02 | Freewheel前端工程化的演进和最佳实践

下一篇 16 | 应用状态管理（上）：应用状态管理框架Redux

## 精选留言 (1)

 写留言



01

2022-10-11 来自福建

可能需要deepFreeze。本身存在冻结不应该冻结对象的风险  
preact给自己乃至react 提供了 signals。用到了proxy

