

useSignal()—web框架的未来

原文来自Angularjs大佬[MIŠKO HEVERY](#)，这个概念实际上与这些年前端圈讨论的较多的Reactivity和Fine-Grain Reactivity息息相关。简而言之，signal理念带来了2大好处：

1. 细粒度的响应式（Fine-Grain Reactivity）
2. 更好的开发体验（DX）

`useState()`
+ `useRef()`
+ `useMemo()`
- constant rerendering
= ❤️ `useSignal()`

signal是用来存储应用状态的一种方式，与React里的useState类似。但有一些关键的差异使得 Signal 更有优势。

什么是Signal?

Signal和State最关键的区别在于Signal返回的是一个getter和setter，而非响应式系统返回一个value（以及一个 setter）。注意：一些反应式系统同时返回 getter/setter，有些则作为两个单独的引用返回，但思路是相同的。

`useState()` ⇒ value + setter
`useSignal()` ⇒ getter + setter

State与State的差异

问题在于State这个词实际上是混淆了2个不同的概念：

- **StateReference**。状态引用，顾名思义就是一个指向状态的引用。
- **StateValue**。状态值，则是实际存储在状态引用区域的值。

为什么返回一个getter会比直接返回一个值会好呢？因为通过返回 getter，可以将状态引用的传递与状态值的读取分开。

可以看一个SolidJS的代码示例：

```
export function Counter() {
  const [getCount, setCount] = createSignal(0);
  return (
    <button
      onClick={() => setCount(getCount() + 1)}>
      {getCount()}
    </button>
  );
}
```

- createSignal()：分配了状态存储区并且将其初始化为0
- getCount：是一个对存储区的引用，可以将其进行传递
- getCount()：则用来获取状态值

我不理解！在我看来没啥区别

上面解释了Signal和我们之前熟知的State的差别，但是没有解释为啥我们需要关注它。


Signal是响应式的！这意味着它们需要跟踪谁对状态感兴趣（即订阅），如果状态变了，通知订阅者状态发生了变更。

为了实现这种响应式，Signal需要收集谁对Signal的值感兴趣，怎么收集的呢？它是通过观察Signal的getter是在哪些上下文被调用进而收集到这些信息的。当你通过getter获取Signal的值时，实际上是在告诉Signal这个位置对Signal的值感兴趣。当Signal的值发生变化时，这个位置需要被重新计算，换句话说，**调用Signal的getter创建了一个订阅**。

这也就是为什么传递状态的getter比传递值更为重要，传递状态值并不会给Signal关于这个值实际在哪里用的任何信号，这也就是为什么区分状态引用和状态值在Signal这里非常重要的原因。

作为比较，下面是一个使用Qwik写的例子，注意这里的getter/setter已经被一个单独的、拥有 `.value` 属性（用来表示getter/setter）的值替代，尽管语法上是有区别的，但是内部工作原理是一样的。

```
export function Counter() {  
  const count = useSignal(0);  
  return (  
    <button  
      onClick$={() => count.value++}>  
        {count.value}  
    </button>  
  );  
}
```




看起来和我们平时写React没啥差别，但是重要的区别是，当我们点击按钮，按钮值从0变成1时，框架只需要将文本节点的值从0更新成1，并不会对整个组件重新渲染。能这么做的原因在于，当初次渲染模板的时候，Signal知道count.value仅仅只被这个文本节点所使用。因此，它知道当count发生改变的时候，它只需要更新文本节点，别的都不需要管。

useState()的缺点

我们可以看看React是如何使用useState的以及它的缺点。

```
export function Counter() {  
  const [count, setCount] = useState(0);  
  return (  
    <button  
      onClick={() => setCount(count + 1)}>  
      {count}  
    </button>  
  );  
}
```

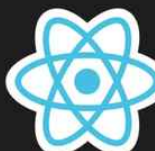


React的useState返回的是一个状态值。这意味着useState对于这个状态值在组件或应用中是如何被使用的没有任何概念。这意味着，当你通过调用 setCount() 通知 React 状态更改时，React 其实不知道页面的哪一部分已更改，因此必须重新渲染整个组件。这在计算上是昂贵的。

useRef()不会触发渲染

React有useRef(), 它和useSignal() 有些类似，但是它不会触发UI重新渲染，下面的例子看起来和useSignal()类似，但是它并不能按预期工作。

```
export function Counter() {  
  const count = useRef(0);  
  return (  
    <button  
      onClick={() => count.current ++}>  
      {count.current}  
    </button>  
  );  
}
```



useRef() 的使用方式与 useSignal() 完全相同，用于传递对状态的引用而不是状态本身。useRef() 缺少的是订阅跟踪和通知。

好处是，在基于Signal的框架中，useSignal() 和 useRef() 是同一件事。useSignal() 可以完成useRef() 的功能以及订阅跟踪。这进一步简化了框架的API。


useMemo()的功能则是内置的

Signal很少需要类似React里memo的功能，因为它们开箱即用的工作量最少。

考虑下面这个例子，有2个计数器和2个组件：

```
export function Counter() {
  console.log("<Counter/>");
  const countA = useSignal(0);
  const countB = useSignal(0);
  return (
    <div>
      <button onClick$={() => countA.value++}>A</button>
      <button onClick$={() => countB.value++}>B</button>
      <Display count={countA.value} />
      <Display count={countB.value} />
    </div>
  );
}

export const Display = component$(
  ({ count }: { count: number }) => {
    console.log(`<Display count=${count}>`);
    return <div>{count}</div>;
  }
);
```



在上面的例子中，当点击一次后，只有其中一个Display组件的文本节点会被更新。未更新的文本节点在初次渲染后不会再次渲染。


```
# Initial render output
<Counter/>
<Display count={0}/>
<Display count={0}/>

# Subsequent render on click
(blank)
```

实际上在 React 中你无法实现同样的目标，因为至少，至少有一个组件需要重新渲染。那么让我们看看如何在 React 中 memo 组件以最大程度地减少重新渲染的次数。

```
export default function Counter() {
  console.log('<Counter/>');
  const [countA, setCountA] = useState(0);
  const [countB, setCountB] = useState(0);
  return (
    <div>
      <button onClick={() => setCountA(countA + 1)}>A</button>
      <button onClick={() => setCountB(countB + 1)}>B</button>
      <MemoDisplay count={countA} />
      <MemoDisplay count={countB} />
    </div>
  );
}

export const MemoDisplay = memo(Display);

export function Display({ count }: { count: number }) {
  console.log(`<Display count=${count}>`);
  return <div>{count}</div>;
}
```



但是，即使使用了 memo，再次点击后，React 仍然会进行如下渲染：

```
# Initial render output
<Counter/>
<Display count={0}/>
<Display count={0}/>

# Subsequent render on click
<Counter/>
<Display count={1}/>
```

没有memo的话，则会像下面这样

```
# Initial render output
<Counter/>
<Display count={0}/>
<Display count={0}/>

# Subsequent render on click
<Counter/>
<Display count={1}/>
<Display count={0}/>
```

这比 Signals 要做的工作要多得多。因此，这就是为什么Signal的工作方式就好像帮你记住了所有内容，而实际上不必自己memo任何内容。

Prop下钻

可以使用一个非常普遍的商品车的例子来看看

```
export default function App() {
  console.log("<App/>");
  const [cart, setCart] = useState([]);
  return (
    <div>
      <Main setCart={setCart} />
      <NavBar cart={cart} />
    </div>
  );
}
```

```
export function Main({ setCart }: any) {
  console.log(`<Main/>`);
  return (
    <div>
      <Product setCart={setCart} />
    </div>
  );
}

export function Product({ setCart }: any) {
  console.log(`<Product/>`);
  return (
    <div>
      <button onClick={() =>
        setCart((cart: any) =>
          [...cart, "product"])}>
        Add to cart
      </button>
    </div>
  );
}
```

```
export function NavBar({ cart }: any) {
  console.log(`<NavBar/>`);
  return (
    <div>
      <Cart cart={cart} />
    </div>
  );
}

export function Cart({ cart }: any) {
  console.log(`<Cart/>`);
  return <div>
    |   |   |   |   Cart: {JSON.stringify(cart)}
    |   |   |   |   </div>;
  </div>;
}
```

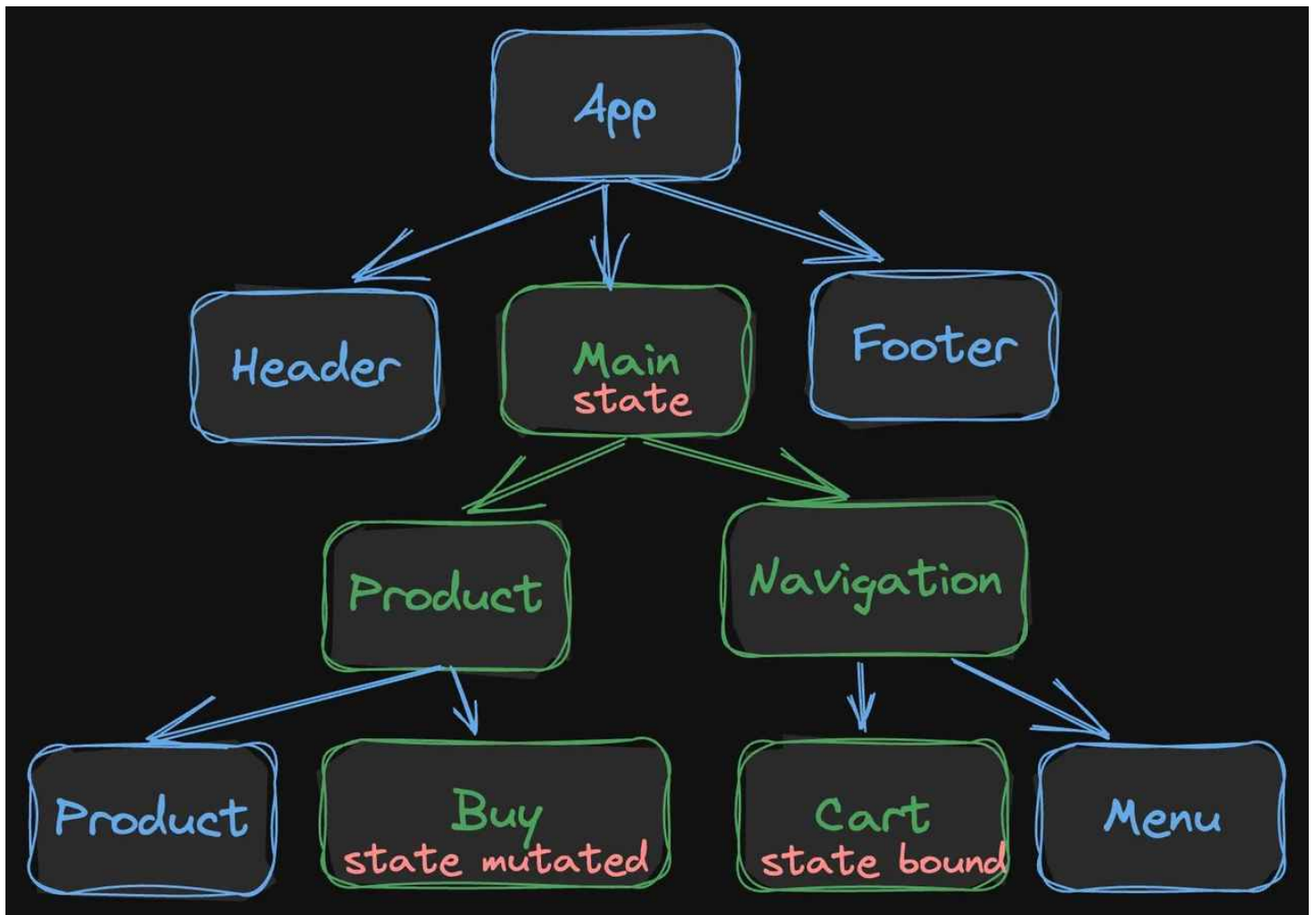


购物车的状态通常会被提升到加购按钮和呈现购物车的位置之间的最高公共父级。由于加购按钮和购物车在 DOM 中相距较远，因此通常这个公共父级非常接近组件渲染树的顶部。在我们的例子中，我们将其称为共同祖先组件。

共同祖先组件有2个分支：

- 其中Main组件分支将setCart往下透传很多层直到购买按钮组件
- NavBar组件则将cart透传很多层直到购物车组件

存在的问题是，每次你点击购买按钮，大部分组件树都将被重新渲染，大体上会是这样子：

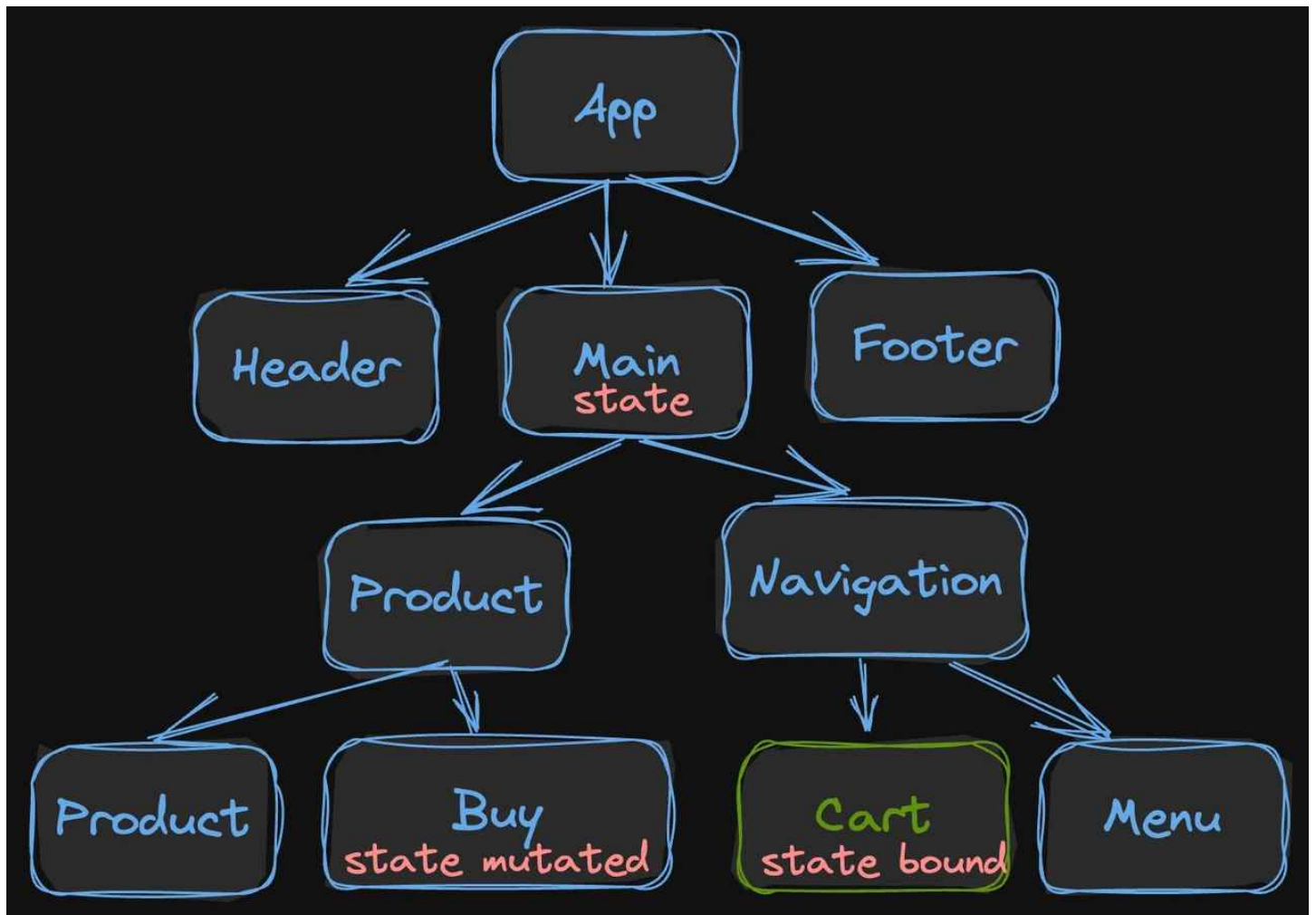


```
# "buy" button clicked  
<App/>  
<Main/>  
<Product/>  
<NavBar/>  
<Cart/>
```

如果你使用memo进行优化，可以避免setCart往下透传导致的渲染，但是没法避免cart变化导致的渲染：

```
# "buy" button clicked  
<App/>  
<NavBar/>  
<Cart/>
```

但是如果用Signal实现的话，会是下面这样：



```
# "buy" button clicked  
<Cart/>
```

这将大大减少需要执行的代码量。

哪些框架支持了这样的特性？

一些主流的框架（如Vue、Preact、Solid、Qwik）都已经支持了。



useSignal()



shallowRef()



现在，Signal并不是什么新鲜事。在此之前它们已经存在于 Knockout 和可能的其他框架中。不同的是，Signal近年来通过巧妙的编译器技巧和与 JSX 的深度集成极大地改进了它们的 DX，这使得它们非常简洁并且使用起来很愉快——而且这部分是真正的新部分。

总结

Signal是在应用程序中存储状态的一种方式，类似于 React 中的 `useState()`。然而，关键的区别在于 Signal 返回一个 `getter` 和一个 `setter`，而非响应式系统仅返回一个值和一个 `setter`。

这很重要，因为 Signal 是响应式的，这意味着它们需要跟踪谁对状态感兴趣并通知订阅者状态更改。这是通过观察调用状态获取器的上下文来实现的，这会创建订阅。

相比之下，React 中的 `useState()` 仅返回状态值，这意味着它不知道状态值如何使用，并且必须重新渲染整个组件树以响应状态更改。

近年来，Signal 已经达到了新阶段，这使得它们并不比传统系统更难使用。因此，我认为你将使用的下一个框架将是响应式的并且基于 Signal 的。

参考

- <https://www.builder.io/blog/usesignal-is-the-future-of-web-frameworks#code-use-ref-code-does-not-render>
- <https://www.builder.io/blog/history-of-reactivity>