08-复杂状态处理：如何保证状态一致性？

你好，我是王沛。今天我们聊聊 React 中的状态管理。

从这节课开始，我们就进入到了实战篇的训练。React Hooks 中其实有一些通用原则和常见设计模式，所以我设计了几个典型的业务场景，这样你就可以对这些原则和模式有一个具体的印象，之后在遇到类似场景时，也能从容应对。

今天我们先从状态一致性这个需求开始讲起。我在基础篇就反复提到过，React 中 UI 完全是通过状态驱动的。所以在任何时刻，整个应用程序的状态数据就完全决定了当前 UI 上的展现。毫不夸张地说，React 的开发其实就是复杂应用程序状态的管理和开发。因此，这就需要你去仔细思考，**该怎么去用最优、最合理的方式，去管理你的应用程序的状态。**

所以今天这节课我会带你围绕状态一致性这个需求，介绍两个基本原则，它们能帮助我们避免很多复杂的状态管理逻辑，简化应用程序的开发。

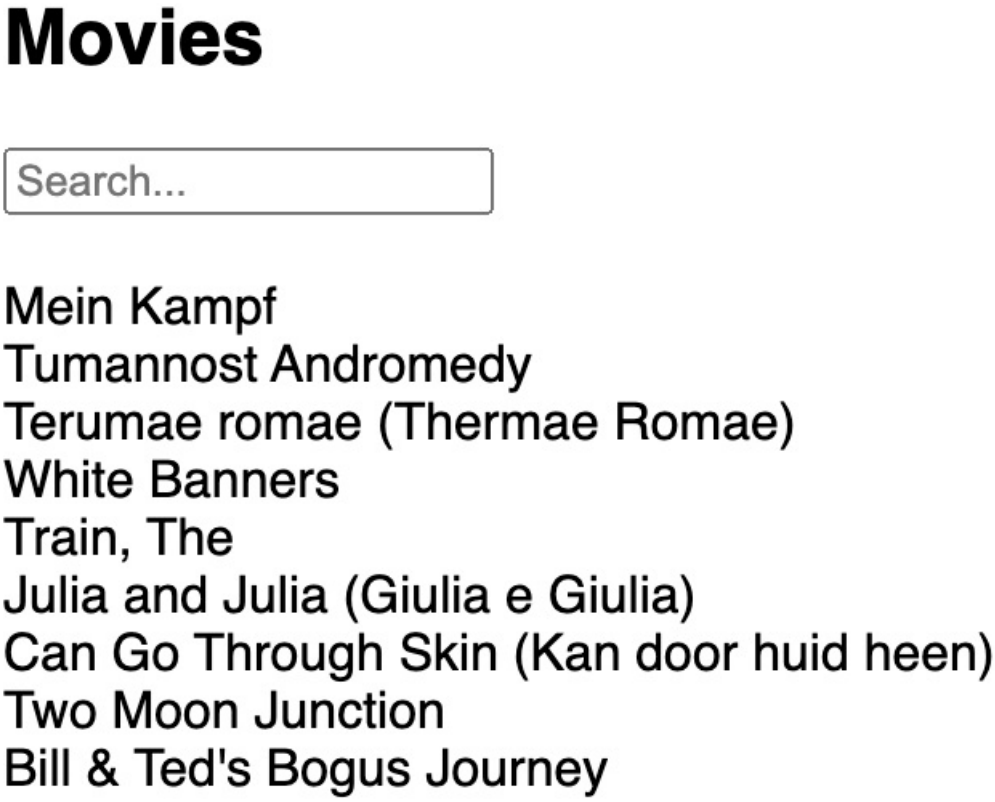
原则一：保证状态最小化

新接触 React 的同学经常会有一个错误的习惯，就是把 State 当变量用，很容易把过多的数据放到 State 里，可以说这是对 State 的一种滥用。

那到底该怎么使用 State 呢？我们需要遵循一个原则，即：**在保证 State 完整性的同时，也要保证它的最小化。**什么意思呢？

就是说，某些数据如果能从已有的 State 中计算得到，那么我们就应该始终在用的时候去计算，而不要把计算的结果存到某个 State 中。这样的话，才能简化我们的状态处理逻辑。

举个例子。你要做一个功能，需要对一个列表的结果进行关键字搜索，我们假设是一个显示电影标题的列表，需要能够对标题进行搜索。最终的效果如下图所示：



可以看到，这个功能包括一个搜索框和一个电影标题的列表。那么，在考虑怎么实现这个功能的时候，应该从哪里着手呢？

按照 React 的状态驱动 UI 的思想，第一步就是**要考虑整个功能有哪几个状态**。直观上来说，页面可能包含三个状态：

1. 电影列表的数据：可能来自某个 API 请求；
2. 用户输入的关键字：来自用户的输入；
3. 搜索的结果数据：来自原始数据结合关键字的过滤结果。

那么很多同学这时候就会在组件中去定义这三个状态，一般的实现代码如下：

```
function FilterList({ data }) {
  // 设置关键字的 State
  const [searchKey, setSearchKey] = useState('');
  // 设置最终要展示的数据状态，并用原始数据作为初始值
  const [filtered, setFiltered] = useState(data);

  // 处理用户的搜索关键字
  const handleSearch = useCallback(evt => {
    setSearchKey(evt.target.value);
    setFiltered(data.filter(item => {
      return item.title.includes(evt.target.value)));
    }));
  }, [filtered])
  return (
    <div>
      <input value={searchKey} onChange={handleSearch} />
      {/* 根据 filtered 数据渲染 UI */}
    </div>
  );
}
```

看上去没有太大问题，整段代码也能正常工作。但是如果仔细思考，你会发现其中隐藏的一致性的问题：展示的结果数据完全由原始数据和关键字决定，而现在却作为一个独立的状态去维护了。这意味着你始终要在原始数据、关键字和结果数据之间保证一致性。

在代码中，我们已经做了一部分维护一致性的工作，那就是当关键字变化时，我们会同时更新关键字和最终结果这两个状态，从而让这两个状态始终保持一致。

不过还有部分一致性的工作没有被考虑到，那就是如果原始数据 data 属性变化了，最终的结果却没有使用新的数据。

这个时候你可能就又会问了：我在处理关键字变化的同时，再处理一下 data 属性变化的场景，这样不就可以保证三个状态的一致性了吗？比如再加上下面这段代码。

```
function FilterList({ data }) {
  // ...
  // 在 data 变化的时候，也重新生成最终数据
  useEffect(() => {
    setFiltered(data => {...})
  }, [data, searchKey])
  // ...
}
```

现在，我们终于能够保证三个状态的一致性了，整个搜索列表也能正常工作了！但是我们在获得一些成就感的同时，是不是也有一些小小的抱怨：这状态管理确实还挺复杂的，需要写这么多的逻辑来保证一致性，从而让功能正常工作。

那么，我想说的是，这种复杂性其实完全不需要。因为复杂性的根源就在于没有遵循状态最小化的原则，而是设计了一个多余的状态：**过滤后的结果数据**。由于这个结果数据实际上完全由原始数据和过滤关键字决定，那么我们在需要的时候每次重新计算得出就可以了。

那时候你可能又有疑问了，如果每次都计算，不是会有性能问题吗？其实在第4讲我已经提到，React 提供的 useMemo 这个 Hook 正是为了解决这个问题，可以缓存计算的结果。所以实现的代码可以修改如下：

```
import React, { useState, useMemo } from "react";

function FilterList({ data }) {
  const [searchKey, setSearchKey] = useState("");

  // 每当 searchKey 或者 data 变化的时候，重新计算最终结果
  const filtered = useMemo(() => {
    return data.filter((item) =>
      item.title.toLowerCase().includes(searchKey.toLowerCase())
    );
  }, [searchKey, data]);

  return (
```

```
    <div className="08-filter-list">
      <h2>Movies</h2>
      <input
        value={searchKey}
        placeholder="Search..."
        onChange={(evt) => setSearchKey(evt.target.value)}
      />
      <ul style={{ marginTop: 20 }}>
        {filtered.map((item) => (
          <li key={item.id}>{item.title}</li>
        ))}
      </ul>
    </div>
  );
}
```

可以看到，除了通过属性传递进来的 data 状态，我们只定义了一个 searchKey 这个状态。然后通过计算的方式，就可以得到最终需要展现的结果。这样，状态的一致性就得到了天然的保证。你看，通过使用状态最小化的原则，管理就变得非常简单了。

虽然这是一个比较简单的例子，但是在实际开发的过程中，很多复杂场景之所以变得复杂，如果抽丝剥茧来看，你会发现它们都有**定义多余状态现象**的影子，而问题的根源就在于**它们没有遵循状态最小化的原则**。

所以我们在定义一个新的状态之前，都要再三拷问自己：**这个状态是必须的吗？是否能通过计算得到呢？**在得到肯定的回答后，我们再去定义新的状态，就能避免大部分多余的状态定义问题了，也就能在简化状态管理的同时，保证状态的一致性。

原则二：避免中间状态，确保唯一数据源

上面的例子其实定义的多余状态比较明显，但在有的场景下，特别是原始状态数据来自某个外部数据源，而非 state 或者 props 的时候，冗余状态就没那么明显。这时候你就需要准确定位状态的数据源究竟是什么，并且在开发中确保它始终是唯一的数据源，以此避免定义中间状态。

还是拿刚才讲的可搜索电影列表的例子来说，我们需要在用户体验上做一个改进，要让搜索的结果做到可分享。这个功能就类似 Baidu 这样的搜索引擎，通过一个链接就能分享搜索结果。比如说，你通过“http://www.baidu.com/s?wd=极客时间”就可以看到极客时间的搜索结果。

想象一下，如果搜索引擎没有这个功能，那使用起来会有多么不方便。每次要分享一个搜索结果，都必须告诉别人关键字是什么，让他/她自己打开 Baidu 去搜索。所以将关键字放到 URL 中也是实际开发中经常遇到的一个需求。

那么要实现这个功能，我们就需要让 URL 中包含搜索关键字的信息，这样任何人用这个 URL ，就都能看到和我一样的搜索关键字和结果了。

在考虑这个功能实现，尤其是基于已有功能做改进的时候，通常来说，直观的思路都是：首先把 URL 上的参数数据保存在一个 State 中，当URL 变化时，就去改变这个 State。然后在组件中再根据这个 State 来实现搜索的业务逻辑。

按照这个思路来改进电影列表的话，我们就可以用类似下面的代码来实现：

```
// getQuery 函数用户获取 URL 的查询字符串
import getQuery from './getQuery';
// history 工具可以用于改变浏览器地址
import history from './history';

function SearchBox({ data }) {
  // 定义关键字这个状态，用 URL 上的查询参数作为初始值
  const [searchKey, setSearchKey] = useState(getQuery('key'));
  // 处理用户输入的关键字
  const handleSearchChange = useCallback(evt => {
    const key = evt.target.value;
    // 设置当前的查询关键状态
    setSearchKey(key);
    // 改变 URL 的查询参数
    history.push(`/movie-list?key=${key}`);
  })
  // ....
  return (
    <div className="08-search-box">
      <input
        value={searchKey}
        placeholder="Search..."
        onChange={handleSearchChange}
      />
      {/* 其它渲染逻辑*/}
    </div>
  );
}
```

可以看到，组件的核心逻辑基本没变，只是做了两个小的变化：

1. 把 URL 上的查询参数作为关键字的默认值；
2. 当用户输入搜索关键字时，我们不但更新了内部 State，同时还改变了 URL 的查询参数。

通过这两个小的变化，我们就实现了搜索结果的可分享。看上去似乎没有什么问题：保证了关键字状态，还有 URL 参数的一致性。但是正如我刚才强调的，一旦涉及到主动保持一致性的逻辑，我们就要考虑状态是否真的有必要。

所以我们再仔细思考下，就会发现上面的逻辑是有漏洞的。因为从 URL 参数到内部 State 的同步只有组件第一次渲染才会发生，而后面的同步则是由输入框的 onChange 事件保证的，那么一致性就很容易被破坏。

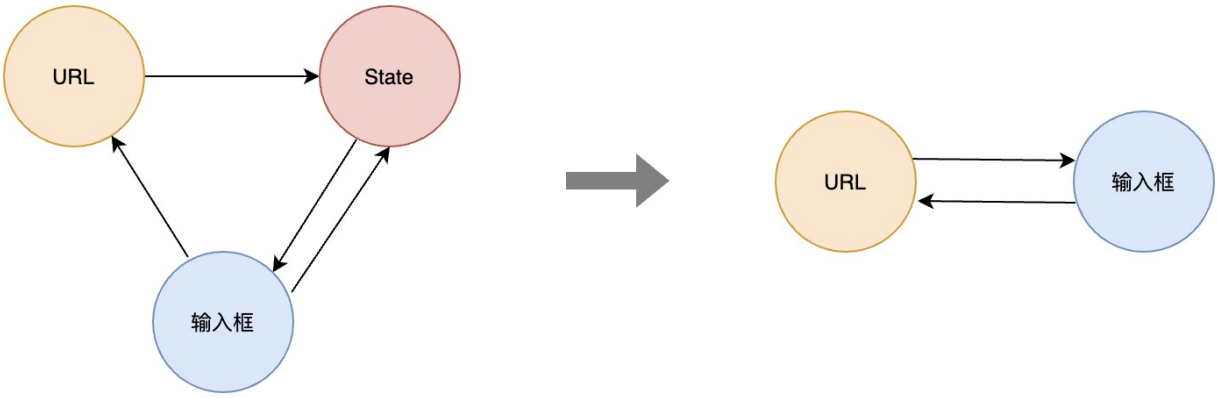
也就是说，如果 URL 不是由用户在组件内搜索栏去改变的，而是其它地方，比如说组件外的某个按钮去触发改变的，那么组件由于已经渲染过了，其实内部的 searchKey 这个 State 是不会被更新的，一致性就会被破坏。

要解决这个问题，一个比较容易想到的思路就是**我们要有更加完善的机制，让在 URL 不管因为什么原因而发生变化的时候，都能同步查询参数到 searchKey 这个 State**。但是如果沿着这个思路，那么状态管理就会一下子变得非常复杂。因为我们需要维护三个状态的一致性。

其实，从根源上来说，产生这个复杂度的问题在于我们定义了 searchKey 这样一个多余的中间状态，而且

这个 searchKey 状态来源于两个数据源：一是用户输入；二是 URL 上的参数。这就导致逻辑非常复杂。

那么如果我们遵循唯一数据源这个原则，把 URL 上的查询关键字作为唯一数据源，逻辑就会变得简单了：只需要在用户输入查询关键字时，直接去改变 URL 上的查询字符串就行。这个转变可以用下面的图做一个对比，你会更直观地理解：



通过对比可以看到，左边引入了一个多余的 State 作为关键字这个状态，而为了保证一致性，就需要很多复杂的同步逻辑，比如说以下几点：

- URL 变化时，同步查询关键字到 State；
- State 变化时，同步查询关键字到输入框；
- 用户在输入框输入的时候，同步关键字到 URL 和 State。

但是，在去掉多余的 State 后，我们就只需要在输入框和 URL 的查询参数之间做一个同步。那么实现的代码可以简化如下：

```
import React, { useCallback, useMemo } from "react";
import { useSearchParam } from "react-use";

function SearchBox({ data }) {
  // 使用 useSearchParam 这个 Hook 用于监听查询参数变化
  const searchKey = useSearchParam("key") || "";
  const filtered = useMemo(() => {
    return data.filter((item) =>
      item.title.toLowerCase().includes(searchKey.toLowerCase())
    );
  }, [searchKey, data]);

  const handleSearch = useCallback((evt) => {
    // 当用户输入时，直接改变 URL
    window.history.pushState(
      {},
      "",
      `${window.location.pathname}?key=${evt.target.value}`
    );
  }, []);
  return (
    <div className="08-filter-list">
      <h2>Movies (Search key from URL)</h2>
```

```
      <input
        value={searchKey}
        placeholder="Search..."
        onChange={handleSearch}
      />
      <ul style={{ marginTop: 20 }}>
        {filtered.map((item) => (
          <li key={item.id}>{item.title}</li>
        ))}
      </ul>
    </div>
  );
}
```

可以看到，当用户输入参数的时候，我们是直接改变当前的 URL，而不是去改变一个内部的状态。所以当 URL 变化的时候，我们使用了 useSearchParam 这样一个第三方的 Hook 去绑定查询参数，并将其显示在输入框内，从而实现了输入框内容和查询关键字这个状态的同步。

从本质上来说，这个例子展示了确保状态唯一数据源的重要性。我们是直接将 URL 作为唯一的数据来源，那么状态的读取和修改都是对 URL 直接进行操作，而不是通过一个中间的状态。这样就简化了状态的管理，保证了状态的一致性。

实战演练：创建自定义受控组件

在前面两个例子中，你看到了状态管理的两个很重要的原则，一个是确保状态最小化，另一个则是找到正确的状态来源，并直接使用这个来源，避免中间状态。那么接下来我再通过一个日常开发中非常常见的例子，来帮助你理解和掌握这两个原则的实际应用。

这个例子就是创建一个受控表单组件。比如，一个用于输入价格的表单组件，需要用户既能输入价格的数量，还能选择货币的种类，最终的效果类似下面这个图：

首先我要解释下什么是受控组件。在 React 中，对表单组件的处理可以分为两种，受控组件和非受控组件：

1. 受控组件：组件的展示完全由传入的属性决定。比如说，如果一个输入框中的值完全由传入的 value 属性决定，而不是由用户输入决定，那么就是受控组件，写法是：< input value={value} onChange={handleChange} />。这也是为什么只给 < input/> 传了一个 value 值但是没有传 onChange 事件，那么键盘怎么输入都没有反应。
2. 非受控组件：表单组件可以有自己的内部状态，而且它的展示值是不受控的。比如 input 在非受控状态下的写法是：< input onChange={handleChange}/>。也就是说，父组件不会把 value 直接传递给 input 组件。

在日常开发中，大部分的表单元素其实都是受控组件，我们会通过外部的状态完全控制当前组件的行为。

那么对于这个例子，价格输入框作为一个受控组件，它需要定义两个属性：value 和 onChange。这样它就和一个普通的表单元素具有相同用法了。其中 value 的值是一个对象，同时包含数值和货币两个属性，比

如：

```
{
  amount: 0,
  currency: 'rmb',
}
```

那现在我们就来看如何实现这个受控组件。在这里我就不再演示错误的写法是什么样的了，而是直接给你看正确的实现方式。

在实际项目中，我经常看到很多同学会把这个简单的功能做得逻辑非常复杂，甚至还不断出现 Bug，总不能保证 UI 和数据的一致性。这其实都是因为没有仔细思考状态的准确来源是什么，以及是否定义了多余的状态。

我们先直接来看正确的实现：

```
import React, { useState, useCallback } from "react";

function PriceInput({
  // 定义默认的 value 的数据结构
  value = { amount: 0, currency: "rmb" },
  // 默认不处理 onChange 事件
  onChange = () => {}
}) {
  // 定义一个事件处理函数统一处理 amount 或者 currency 变化的场景
  const handleChange = useCallback(
    (deltaValue) => {
      // 直接修改外部的 value 值，而不是定义内部 state
      onChange({
        ...value,
        ...deltaValue
      });
    },
    [value, onChange]
  );
  return (
    <div className="exp-02-price-input">
      {/* 输入价格的数量 */}
      <input
        value={value.amount}
        onChange={(evt) => handleChange({ amount: evt.target.value })}
      />
      {/* 选择货币种类*/}
      <select
        value={value.currency}
        onChange={(evt) => handleChange({ currency: evt.target.value })}
      >
        <option value="rmb">RMB</option>
        <option value="dollar">Dollar</option>
      </select>
    </div>
  );
}
```

可以看到，这个自定义组件包含了 input 和 select 两个基础组件，分别用来输入价格数量和选择货币。在它们发生变化的时候，直接去触发 onChange 事件让父组件去修改 value 值；同样的，它们自己显示的值，则完全来自于传递进来的 value 属性。所以这其中的思考逻辑在于：

1. 避免多余的状态：我们不需要在 PriceInput 这个自定义组件内部，去定义状态用于保存的 amount 或者 currency。
2. 找到准确的唯一数据源：这里内部两个基础组件的值，其准确且唯一的来源就是 value 属性，而不是其它的任何中间状态。

因此，通过这样的做法，整个自定义的组件逻辑就变得非常简单，甚至不需要任何内部的状态，就实现了这样一个非常常见的需求。

而在我看到的实际项目代码中，发现很多同学都习惯于用多余的内部状态去分别保存 amount 和 currency，然后再和外部传进来的 value 属性进行同步，以此来保证一致性，这就造成了状态逻辑的复杂。所以我们在做类似功能的时候，一定要避免这种不合理的做法，尽量用最简洁的逻辑去实现需要的功能。

小结

好了，这节课的内容就是这些，我简单小结一下。你可以把 React 的开发看作是复杂状态的管理和维护。那么为了保证状态的一致性，我们就一定要简化状态处理的逻辑。其中有两个重要的原则需要遵循：

- 一个是状态最小化原则，也就是说要避免冗余的状态；
- 另一个则是唯一数据源原则，避免中间状态。

所以，在任何时候想要定义新状态的时候，都要问自己一下：这个状态有必要吗？是否能通过计算得到？是否只是一个中间状态？只有每次都仔细思考了，才能找到需要定义的最本质的状态。然后围绕这个最本质的状态去思考某个功能具体的实现，从而让 React 的开发更加简洁和高效。

这节课所有的示例代码，你可以通过 codesandbox 查看：https://codesandbox.io/s/react-hooks-course-20vzg.

思考题

在第二个在URL 中包含查询关键字的例子中，我们用到了 userSearchParams 这样一个第三方的 Hook，用于绑定 URL 上的查询字符串参数。如果让你实现这个 Hook，你会怎么做呢？

欢迎在评论区写下你的思考和想法，我会和你交流讨论。如果今天的实战演练让你有所收获，也欢迎把课程分享给你的同事、朋友，我们共同进步！

精选留言：

- Geeker 2021-06-10 07:26:30
  状态最小化原则直接影响了代码的复杂度 [1赞]

  作者回复2021-06-11 13:22:10
  可以说，直接降低了代码复杂度~

- 守望 2021-06-10 21:41:25
  这是不是和单向数据流差不多呀！！！
  数据流向总是一条线，不要开新分支

  作者回复2021-06-11 13:18:14
  虽然看上去有点像，但不太一样哦~

- 傻子来了快跑丶 2021-06-10 21:12:51
  // 每当 searchKey 或者 data 变化的时候，重新计算最终结果
  老师这个地方有问题吧，一般我们是通过关键字去拿接口的data数据，你这个demo data数据并不是通过keyword获取过来的，而是直接传入的一个data，是有问题的，应该通过状态提升，也就是
  const filtered = useMemo(() => { return data.filter((item) => item.title.toLowerCase().includes(searchKey.toLowerCase())); }, [searchKey, data]);
  这段代码应该在父组件中，拿到data数据之后再传进去

  作者回复2021-06-11 13:20:38
  数据处理为什么要在父组件呢?

- Jerryz 2021-06-10 18:14:04
  默认 history.pushState 和 history.replaceState 都没有对应的监听事件。react-use patch 了history 对象。

  作者回复2021-06-11 13:21:16
  👍

- Isaac 2021-06-10 10:27:59
  思考题回答：
  由于 hisgory.pushState 不会触发页面重新渲染，也不会导致组件更新，所以，默认的 userSearchParams 只会获取第一次的 URL 上的查询字符串。因此，为了解决这个问题，可以通过监听 pushstate、replaceState 等事件，对状态进行同步。

  其实去阅读 react-us 的源码实现，也是采用了这种办法。
  https://github.com/streamich/react-use/blob/90e72a5340460816e2159b2c461254661b00e1d3/src/useSearchParam.ts#L8

  作者回复2021-06-12 11:57:52
  赞~ history API 是比较特别的，只能用 patch 的方法来监听 url 变化。

- Isaac 2021-06-10 10:10:29
  老师，文章中的列表筛选的例子，虽然使用 useMemo 可以缓存，但是如果多个组件实例用到，岂不是还是会出现多次计算?

  作者回复2021-06-12 11:58:29
  是的，跨组件的数据共享就需要 Redux 这样的全局状态管理机制了。