答疑解惑02

王沛 2021-07-13

00:00 1.0x

讲述：王沛 大小：11.54M 时长：12:35

你好，我是王沛。实战模块是我们这门课的学习重点，我看到不少同学不仅跟上了学习进度，而且还跟着课程，动手写代码，并把其中讲解的思路应用到了自己的实际项目中。理论结合实践，学以致用，这是一种很有效的学习方法。

与此同时，我也看到有同学在留言区提出了很多有价值的问题。所以这节课呢，我会针对你在实战模块提出的一些具有代表性的问题，进行一个集中的答疑。一方面算是对课程内容做一个有针对性地补充，另外一方面也希望能对更多的同学有所启发和帮助。

第 9 讲

题目 1：article?.userId 和 article&&article.userId 的作用是一样的吗？第一次见这种写法，感觉好简洁。

回答：虽然这是一个 JS 的语法问题，但是因为是一个新语法，所以既然有同学问了，就拿出来讲一下。简单来说， “?.” 是一个名为 optional chaining 的新语法，是刚刚进入 ECMAScript 的标准。借助于 Babel 我们现在可以放心使用。

article?.userId 和 article && article.userIde 这两种写法功能是基本等价的，就是判断 article 是否存在，如果存在则获取 userId 属性，否则就是 undefined。这样的话可以避免 JS 运行时的报错。唯一的一点区别在于，后者 && 的写法其实如果 article 为 null 或者 undefined 或者 0 等 falsy 的值时，会返回这个 falsy 的值本身，比如 null，undefined 或者 0。虽然这在大多数情况下是不用考虑这种差别的。

很显然，optional chaining 这种语法更简洁语义也更明确。下面的代码展示了它的用法：

```
// 静态属性
const c = foo?.bar?.c;
// 动态属性，数组属性
const a = obj?.[key].a;
// 方法调用
obj.getSomething?.();
```

题目 2：对于 loading 和错误处理，我们项目是在做全局处理的，且 loading 是通过 redux 管理的。那么这种做法，跟这节课所讲的方法相比，究竟应该使用哪种方法呢？

回答：两者不冲突，在 Hooks 中课程里是用的 useState，但是如果 loading 和 error 是在 Redux 中，那么可以用 useSelector 去获取状态。全局处理的好处是方便多组件去重用状态，而且可以避免重复的请求。

比如两个组件都需要去使用 Article 数据，如果用 useState 保存数据，那么两个组件就都有各自的数据。如果放到 redux 中，那么逻辑就可以根据是否已经在 loading 来决定是不是要发起请求，还是等待已有的请求结束。

第 10 讲

题目：HOC 存在的意义在哪里呢？感觉能够用 HOC 的场景，其实都可以用 render props 来替代。而且从逻辑角度讲，render props 的逻辑更清晰，HOC 的使用逻辑则更加冗余。所以实在想不出来有什么场景是一定需要使用 HOC 的。

回答：HOC 如果作为 Controller ，而不是提供额外属性时，其语义是更加清晰的。比如说在第 14 课提到的浮动层，我们希望对话框在不可见时就不执行任何逻辑，那么可以用 HOC 来实现一个这样的 Controller，比如下面的代码：

```
const wrapModal = Modal
  => ({visible, ...props})
  => visible ? <Modal visible {...props} /> : null;
```

对于一个 Modal 组件，当 visible 为 false 时，就直接返回 null 而不是去 render Modal 组件。这里就用 HOC 的模式新建了一个 Controller 性质的组件，来重用这种逻辑。

第 12 讲

题目 3：js-plugin 的用处感觉比较大。但是在 react 组件中， 这样的写法，我确实不太能理解。就像是一个容器组件，通过传入不同的参数，返回不同的组件。因为我使用 ts，感觉这样又少了类型检查。

回答：是的，很类似一个容器组件。但是这个容器组件的内容来自于松耦合的不同部分的内容。不是从容器组件内部依赖其它实现，而是反方向，外部的功能自己提供内容到这个容器组件。对于 TypeScript，确实是不够友好的，因为太动态了。但是其实没有类型检查不会是太大问题，因为内容提供者和消费者两者唯一的连接点是 article.footer 这个扩展点以及其函数签名。当然，如果一定要支持 TypeScript，也是可行的，只是比较繁琐。那就是为每个扩展点提供一个类型定义文件，然后在创建 plugin 时，plugin 对象需要满足所有支持的扩展点的类型定义。

第 13 讲

题目 1：虽然这个 API 只支持通过函数执行进行验证，但是我们很容易进行扩展，以支持更多的类型，比如正则匹配、值范围等等。这个能演示一下吗？

回答：要支持更多类型的验证，就是判断当前提供的 validator 的类型，比如如果发现提供的是正则，就用正则匹配，如果是一个包含了 min，max 属性的对象就进行自动的范围判断，如果是函数，那么就调用函数，比如下面的代码在原有的基础上提供了这几种验证机制的支持：

```
if (validators[name]) {
  const validator = validators[name];
  let errMsg;
  if (typeof validator === 'function') {
    // 如果是函数，就调用函数验证
    errMsg = validator(value);
  } else if (validator.min) {
    // 如果指定了 min，就判断是否小于 min
    if (value < validator.min) errMsg = `数值需要小于 ${value}`;
  } else if (validator.max) {
    // 如果提供了 max，就判断是否大于 max
    if (value > validator.max) errMsg = `数值需要大于 ${value}`;
  } else if (validator.test) {
    // 简单的通过是否有 test 方法来确定是不是正则表达式
    if (!validator.test(value)) errMsg = `${name} 需要匹配正则。`
  }

  setErrors((errors) => ({
    ...errors,
    [name]: errMsg || null,
  }));
}
```

这样我们就支持了多种形式的验证，但其实只要提供了函数类型，就能够支持任意形式的同步验证了。加入更多类型只是为了方便这个 Form 机制的使用。

第 14 讲

题目 1：NiceModal 里面有 const modal = useNiceModal(id); 在 MyModalExample 也用了 const modal = useNiceModal("my-modal"); 用了两次 useNiceModal，返回值的两个 modal 应该是不同的对象吧？ modal 里面的 hide、show 函数应该也是不一样的，对吗？不过数据都存在 const store = createStore(modalReducer) 的 store 里了，是全局的。

回答：是的，两个 modal 是不同的对象，show，hide 函数也不一样。useModal 返回的只是用于控制某个对话框的一个 handler。这里的关键在于要理解自定义 Hook 就相当于一个自定义组件，内部使用的任何其它 Hooks，比如 useState，useEffect 都是和外层组件毫无关系的。他们都是存在于自己的作用域里，通过参数和返回值和调用者交互。所以，要实现 Hooks 之间的状态共享，仍然需要 Context，Redux 等全局的状态管理机制。

题目 2：在课程中，我们使用的是 Redux 来管理所有对话框的所有状态。但有时候你的项目并不一定使用了 Redux，那么我们其实也可以使用 Context 来管理对话框的全局状态。那么请你思考一下，如果基于 Context ，应该如何实现 NiceModal 呢？

回答：在基础篇中你曾看到过 useContext 的用法，它也是实现全局状态管理的一个机制。但是，当时并没有提到 useReducer 这个从 Redux 借鉴过来的 API。没有介绍的原因其实在于我个人觉得 useReducer 其实并不常用，因为通常来说我们使用 Redux 等独立的框架会更加方便，因为 API 更加丰富，以及生态更完善，比如 Redux 的一些中间件等等。

但是，在这道题目中，虽然可以不用 useReducer，但是如果使用的话，是可以几乎完全重用课程中例子的代码的。因为 Redux 和 useReducer 的 reducer 和 action 的实现是完全兼容的。这也是为什么很多人说，Context + useReducer 是几乎可以替代 Redux 的。下面的代码展示了使用 Context 的实现：

```
import React, { useContext, useReducer } from 'react';


// 使用一个 Context 来存放 Modal 的状态
const NiceModalContext = React.createContext({});
// modalReducer 和 Redux 的场景完全一样
const modalReducer = ...
function NiceModalProvider({ children }) {
  const [modals, dispatch] = useReducer(modalReducer,{});
  return (
    <NiceModalContext.Provider value={{ modals, dispatch }}>
      {children}
    </NiceModalContext.Provider>
  );
};
```

因为使用了 Context，所以需要在整个应用的根节点创建 Context 并提供所有对话框的状态管理机制。这里也看到了一个非常关键的组合，就是使用了 useReducer 来管理一个复杂的状态，然后再将这个状态作为 Context 的 value。需要注意的是，useReducer 管理的仍然是一个组件的本地状态，和 useState 类似，但其实单个组件很少有状态需要 useReducer 这样的高级功能来管理，所以一般其实 useReducer 都是和 Context 一起使用的，去管理一个更大范围的数据状态。

第 15 讲

题目 1：原文写： “如果你的事件处理函数是传递给原生节点的，那么不写 callback，也几乎不会有任何性能的影响。” 为什么这么说呢？是不是因为原生节点，本来就要不断渲染呢？

回答：要回答这个问题，关键还在于理解在 useCallback 究竟是为了解决什么问题。问题的本质在于，React 组件是通过引用比较来判断某个值是否发生了变化，如果变化了，那么组件就需要重新渲染，以及虚拟 DOM 的 diff 比较。那么，如果每次传过去不同的函数，即使这些函数的功能完全一样，那也会导致组件被刷新。所以，useCallback, useMemo 就是通过缓存上一次的的结果来确保如果功能没变，那么就使用同样的函数，来避免重新渲染。

所以，useCallback，useMemo 只是为了避免 React 组件的重复渲染而导致的性能损失。而对于原生的节点，比如 div, input 这些，它们已经是原子节点了，不再有子节点，所以不存在重复刷新带来的性能损失。

第 16 讲

题目 1：对于 react-loadable 和 Service Worker 有两点比较疑惑的地方。

（1）react-loadable 和 React.lazy() 的使用场景有什么不一样吗？还是说，使用 react-loadable 的地方都可以使用 React.lazy() 代替？

回答：确实，两者确实可以解决一样的问题，毕竟 import() 这个语句是由 Webpack 来处理，最终实现分包的。所以结合 React.lazy 和 Suspense 可以实现 react-loadable 的功能。但是呢，react-loadabale 提供了更丰富的 API，比如可以设置如果加载事件小于 300 毫秒，就不显示 loading 状态。这在使用 service worker 时可以让用户感知不到按需加载的存在，体验更好。因为有研究表明，如果加载时间本来很短，你却一闪而过一个 loading 状态，会让用户觉得时间很长。另外就是，react-loadable 提供了服务器端渲染的支持，而 React.lazy 是不行的。

（2）在 Service Worker 中使用 Cache Storage 来缓存静态资源，是否有容量大小的限制呢？如果是在缓存 svg、png 等其他格式的静态资源的时候，是否有什么限制呢？

回答：Cache 的大小并没有统一的规定，各个浏览器会提供不同的 size，但一个共识是根据当前磁盘剩余大小去调整，从而决定是否能继续存储数据。但是，无论 Size 是多少，我们都需要及时清除不需要的数据，以及一定要处理存储失败的场景，转而使用服务器端的数据。从而保证即使 Cache 满了，也不影响功能的使用。

给文章提建议