

17 | 应用状态管理（下）：该用React组件状态还是Redux？

2022-10-08 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 14:50 大小 13.55M



你好，我是宋一玮，欢迎回到 React 应用开发的学习。

上节课我们学习了应用状态对于 JS 前端应用的重要意义，也学习了以 Redux 为代表的状态管理框架，介绍了 Redux 的核心概念 `action`、`reducer`、`store`，以及它单向数据流的本质。从使用角度，我们介绍了 Redux 封装库 Redux Toolkit 的用法，强调了它对 Redux 开发的简化。

不知你发现没有，上节课除了作为扩展内容的 MobX 和 XState，我们完全没有用到 React。请你放心，当然不是我忘了这个专栏的主题，而是我希望你能以更纯粹的视角，去了解应用状态管理这个领域知识，不会因为 React 的概念导致先入为主。

这节课，我们会把 Redux 与 React 结合起来使用，看看它能为 React 的状态管理带来什么好处，同时也要探讨什么时候该用 Redux，什么时候用 React 内建的 `state`，更或者，是否可以混用两种状态管理。

React 应用中有哪些状态？

我们在开发 React 应用时，会用到各种状态，大致可以分类成三种：业务状态、交互状态以及外部状态。

业务状态是指与业务直接相关的状态，这些状态理论上剥离 UI 也可以使用，比如在单元测试中、Node.js 环境中等等。

举个具体的例子：oh-my-kanban 中的 todoList、ongoingList、doneList，用于保存看板卡片的列表，都可以增删，这些都是 oh-my-kanban 的核心业务，那么它们就属于业务状态。

假设我们非要为 oh-my-kanban 提供一套命令行下的管理工具，那么这些状态和它们相关的逻辑是可以复用的，比如下面这几行命令：

 复制代码

```
1 ohmykanban list --column=ongoing
2 ohmykanban add --column=todo '开发任务-5'
3 ohmykanban remove --column=done '测试任务-2'
```

另外，在大中型 React 项目中的用户权限信息，也经常被存入前端状态中，方便前端逻辑判断某个功能模块是否可以对当前用户开放，或是可见但只读，又或者，直接隐藏起来。

当然，考虑到系统整体安全性，当服务器端接收到用户从浏览器端发起的请求时，仍然要验证用户权限，这类用户权限状态也属于业务状态。

交互状态（也称作 UI 状态），是与用户交互相关的状态，主要控制着用户与应用的交互过程，用于提升用户体验。

比如 oh-my-kanban 中的 isLoading、showAdd，分别控制着是否显示“读取中”占位提示，和是否显示“创建新卡片”的卡片，它们就属于交互状态。

还有，当大中型 React 项目中功能比较多时，常用到的 Tab 标签页，也常用诸如 currentTab 这样的交互状态来记录哪个是当前 Tab。

为什么说交互状态用于提升用户体验呢？请你想象一个极端的设计，从一个 React 应用中删除所有交互状态，包括：

- `isLoading` 不要了，列表拿到数据时会跳一下，还好吧；
- `showAdd` 不要了，那“添加新卡片”默认就一直展示吧，也不是不能用；
- `currentTab` 不要了，所有 Tab 下的页面内容都一次性展示出来，那.....这还能称作是现代前端应用吗？

作为一个**优秀的前端工程师**，**开发出优秀的用户体验是你的职责，也是你的骄傲**，我由衷希望你在这点上不要妥协。

再来讨论一个复杂的问题，🔗[第 11 节课](#)我们学习过受控组件和表单，那么表单状态算是业务状态还是交互状态呢？我认为需要分情况讨论，我们一起来看看。

表单状态属于交互状态：

- 表单状态由若干受控组件状态组成，用户在使用这些受控组件输入文本、选取下拉框时，产生的状态变更主要还是交互行为，暂时还不具有业务意义；
- 在用户录入表单过程中，如果有针对表单项的验证逻辑，比如“标题不能为空”“密码至少需要包含一个数字”，验证过程会使用表单状态，验证结果也会更新到表单状态中。这个时候验证结果已经具有业务意义了，但整体还是可以看作是交互状态。

表单状态属于业务状态：

- 如果表单提供了一个自动提示的下拉框，根据输入的文本内容去服务器端获取下拉框的列表，这个列表就很难说它不是业务状态了；
- 提交表单时会使用其中各个受控组件的最终状态，这时可以认为它们是业务状态；
- 如果这个表单并不是在新建一条记录，而是在修改一条已有记录，那么来自服务器端，为各个表单项提供的初始值也应该算作业务状态。

还有一个场景，比如一个名为 `num` 的状态数据，在某个组件上需要做一系列比较重的计算才能使用，比如 `fibonacci(num)`，有些开发者为了避免反复计算影响性能，把计算结果保存在了另一个名为 `result` 的 `state` 里。

先不用讨论这个计算结果 **result** 是哪种状态，首先我们需要认识到，这是一个计算值，也可以说是派生值。无论是否使用 **Redux**，我们都值得用单一事实来源原则来审视这个计算值：当原值 **num** 和计算值 **result** 都放在状态里，单从状态层面看，是看不出它们的因果关系的。

更合适的做法是，在状态里只保留原值 **num**，组件里从始至终都基于 **num** 做计算。那性能怎么保证呢？我们在 [第 10 节课](#) 学习过的 **useMemo** 就是干这事的：

```
1 const memoizedResult = useMemo(() => fibonacci(num), [num]);
```

 复制代码

上面讨论了业务状态和交互状态，那**外部状态**是怎么回事？在外部不就意味着与 **React** 无关了吗？我举个例子你可能就理解了：**window.location**。

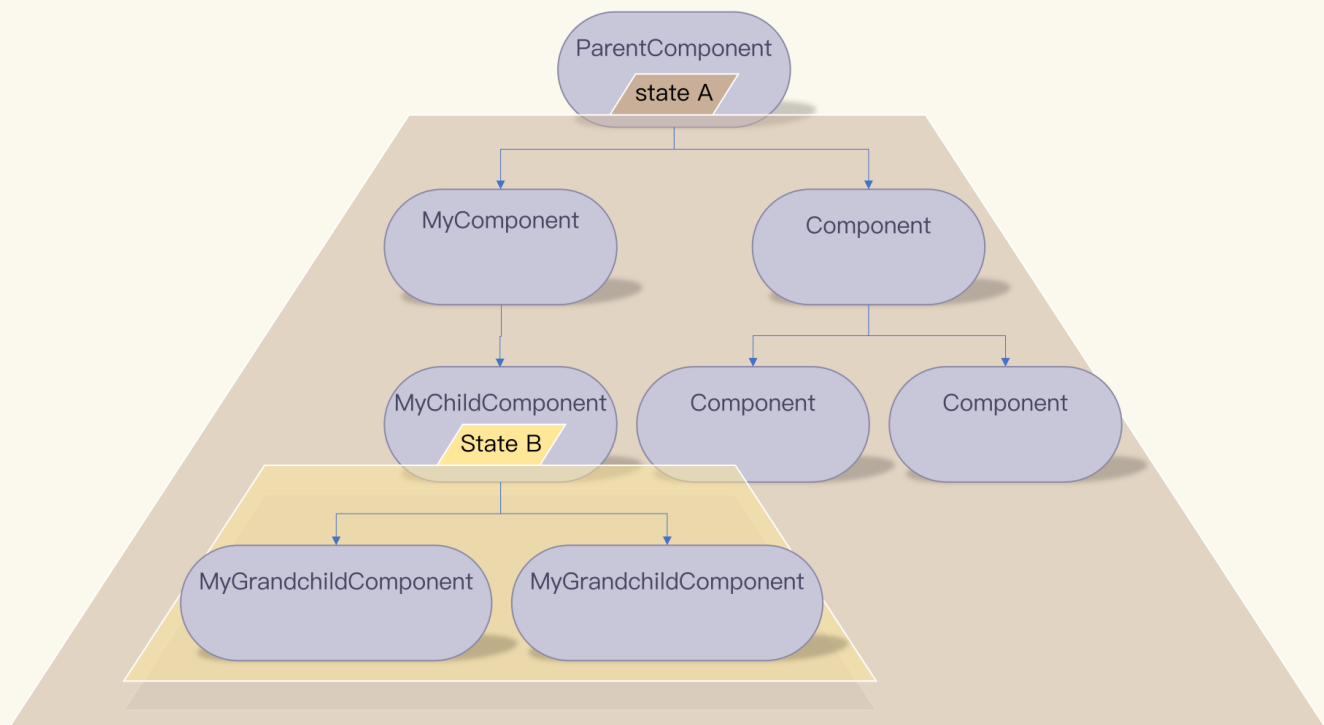
在 **React** 生态中，最常用的前端路由框架就是 **React-Router** 了。**React-Router** 在前端做路由时，会读取 **window.location** 的信息，也会通过浏览器 **History API**，修改 **location** 的 **URL**。这从实际上来看，就成为了 **React** 应用状态的一部分。

其实不仅 **React**，业务状态、交互状态、外部状态的分类对很多其他前端框架也适用。

你这时可能就会有疑问了：“给状态分类是很好，但有什么用？”别急，接下来会用到。

全局状态与局部状态

单看 **API**，**state** 对单个 **React** 组件是私有的，但从单向数据流的角度看，一个组件的 **state** 还可以覆盖到它的所有后代组件。你可以花一分钟研究一下下面这张图，我们再继续往下分析。



ParentComponent 的 state A :

- 可以自用；
- 可以通过 props 传给直接子组件 MyComponent ；
- 也可以通过 props 向下钻取，传给第二层和第三层的后代组件 MyChildComponent 、 MyGrandchildComponent ；
- 更可以通过 context，传给所有的后代组件。

而 MyChildComponent 的 state B 的范围虽然要小很多，但跟 state A 会有一定重合，即 MyChildComponent 、 MyGrandchildComponent 可以同时使用 state A 和 B 的值。

如果 ParentComponent 已经是应用的根组件，那么可以认为 state A 就是全局状态，而 state B 就是局部状态。

当然，全局状态和局部状态是相对而言的。如果你的根组件并未提供状态，而它的唯一子组件提供了状态，那么这个状态也是全局状态。

局部状态也可以通过 [第 13 节课](#) 讲到的**状态提升**这样的开发技巧，根据需要改写为全局状态。

这个时候你可能会有新的疑问：“区分全局与局部状态是很好，但有什么用？”我们接着往下讲。

什么时候使用 Redux？

一般情况下，当你的 React 项目足够小，引入 Redux 的成本要大于收益。只有你**预期项目规模会逐渐增大，或者项目已经是大中型的体量了**，这时可以考虑引入 Redux。Redux 鼓励全局只有单一 store，所以比较适合管理全局状态。

尤其有一种情况，当你发现，你不得不把项目中大部分组件的 state 都提升到根组件上时，全局状态会不断膨胀，那你就有可能亟需引入 Redux 了。

虽然与事实不符，但这里姑且可以认为 Redux 是 React 单向数据流的一层**抽象**。Redux 可以独立于 React 存在，在开发 React 应用时编写的 Redux 代码，与 React 的耦合度比较低，可以独立开发、测试。

给你爆个料，在上节课，Redux 和 Redux Toolkit 的两段样例代码，我就是用 Node 直接跑的：

 复制代码

```
1 npm install redux # 或 npm install @reduxjs/toolkit
2 node index.js
```

将这部分数据流抽象出来后，会降低根组件 state 的复杂度，在编写 Redux 数据流逻辑时，也可以做到与 React 组件的**关注点分离（Separation Of Concerns）**。

另外，**Redux 对状态的变更和读取也是解耦的**。比如用 createSlice 接口创建了 2 个 slice，共同组成 store，reducer 分别更改对应的 state 节点，同时开发者也可以跨 slice 创建 selector 来组合使用多个 state 节点里的数据。

其中 createSelector API 其实就是 reselect 库（[官网](#)），用于创建记忆化的选择器函数：

 复制代码

```
1 import { createSelector } from "@reduxjs/toolkit";
2 // ...
3 const selectBooks = (state) => state.books.allBooks;
```



```

4  const selectFavIds = (state) => state.user.favIds;
5
6  export const selectFavBooks = createSelector(
7    selectBooks,
8    selectFavIds,
9    (books, ids) => {
10      return books.filter(book => ids.includes(book.id));
11    },
12  );

```

下面看看如何在 React 中使用 Redux。

React Redux

我们在 React 中使用 Redux，一般会借助 Redux 官方的 **React 连接器 React Redux**：

 复制代码

```
1 npm install @reduxjs/toolkit react-redux
```

用 **Provider** 组件包住整个应用，传入上节课 Redux Toolkit 样例代码中创建的 store 对象：

 复制代码

```

1  import React from 'react';
2  import ReactDOM from 'react-dom/client';
3  // ...
4  import { Provider } from 'react-redux';
5  import store from './store';
6
7  const root = ReactDOM.createRoot(document.getElementById('root'));
8  root.render(
9    <Provider store={store}>
10      <App />
11    </Provider>
12  );

```

然后在组件中就可以使用这个 store 了：

 复制代码

```

1  import React from 'react';
2  import { useSelector, useDispatch } from 'react-redux';
3  import { addCard, removeCard } from './cardListSlice';

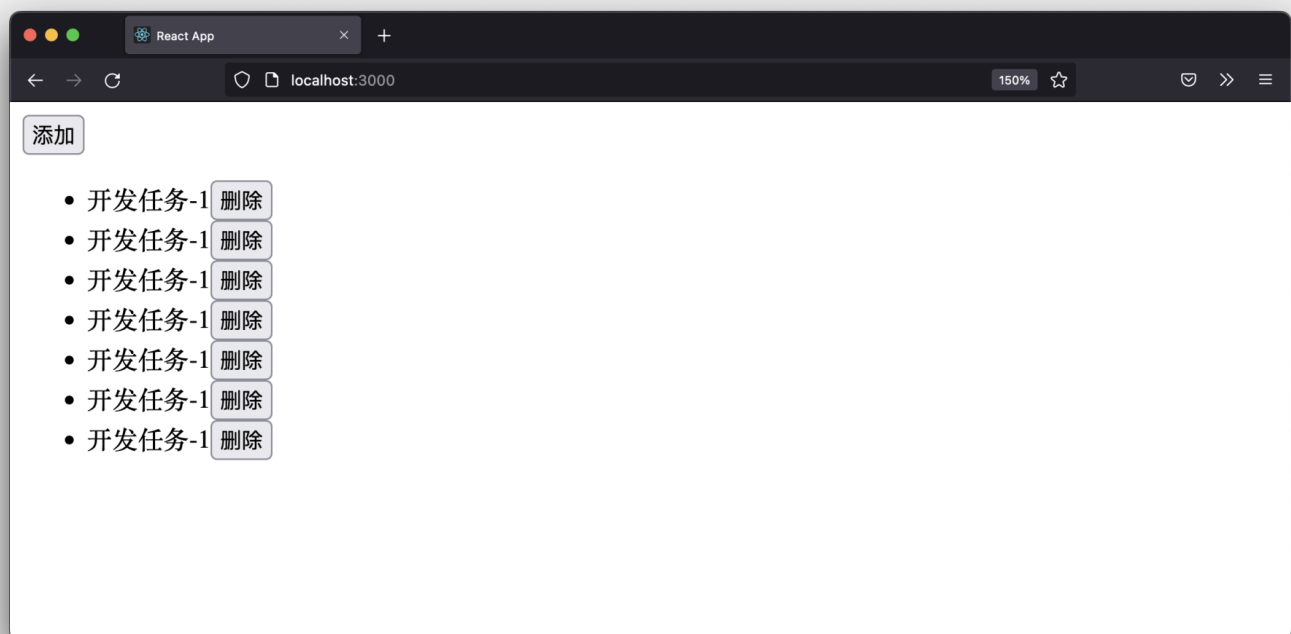
```

```

4 export function CardList() {
5   const cardList = useSelector(state => state);
6   const dispatch = useDispatch();
7
8   return (
9     <div>
10      <button onClick={() => {
11        const payload = { newCard: { title: '开发任务-1' } };
12        dispatch(addCard(payload));
13      }}>添加</button>
14      <ul>
15        {
16          cardList.map(card => (
17            <li key={card.title}>
18              {card.title}
19              <button onClick={() => {
20                dispatch(removeCard({ title: '开发任务-1' }));
21              }}>删除</button>
22            </li>
23          ))
24        }
25      </ul>
26    </div>
27  );
28 }
29

```

相信你用 CRA 很快就能实现这个小 Demo:



这里只是一段临时的例子代码，并不代表我们需要用 Redux 来改写 oh-my-kanban 项目或者你的 yeah-my-kanban 项目。根据前面的什么时候用 Redux 的条件，这两个项目目前还不适合引入 Redux。

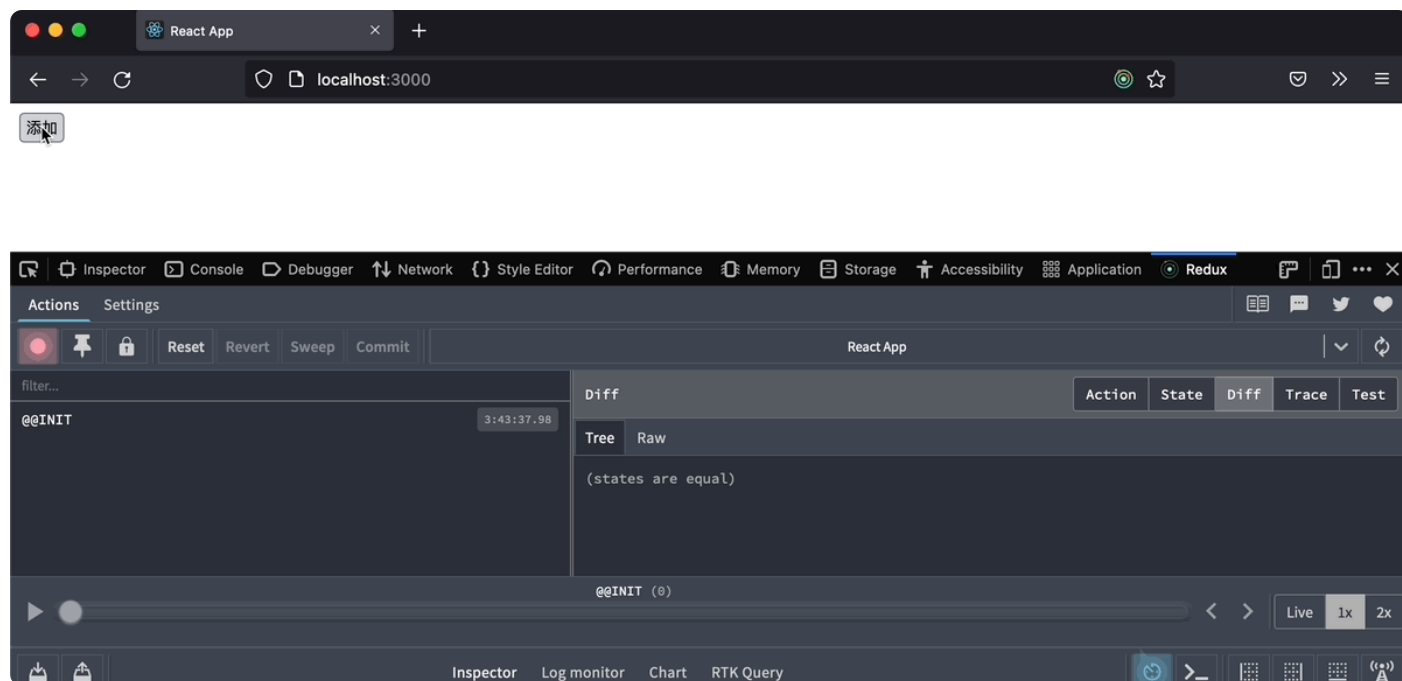
感谢“学习前端 -react”同学在 [🔗 第 12 节课](#) 留言区的提问，我在这里也稍微提一下 React Redux 的原理。

这个库把 Redux 的 store 放到了 context 里，但并没有借助 React 的 context 更新机制来响应 store 内部的更新。在早期版本中，React Redux 提供的高阶组件订阅 store 变化，当有变化时调用组件的 forceUpdate() 方法。

而在新版（v8.0）中，高阶组件使用了 React 的新 Hooks API：useSyncExternalStore，用这个 Hook 返回的 props 来更新被修饰的组件。如果你感兴趣，也可以读一读 React Redux 的 [🔗 发展史](#)。

Redux DevTools 浏览器扩展

经过多年的积累，Redux 开发的生态圈非常丰富，其中我首先推荐的是 Redux 官方推出的 Redux DevTools 浏览器扩展（[🔗 Chrome 扩展链接](#)、[🔗 Firefox 扩展链接](#)），可以用来跟踪调试 state 和 action 包含的数据，还提供了一个很酷的时间旅行功能。



上面的介绍可能引起了你对 Redux 的一些兴趣，但学习归学习，实际项目中还是要认真考虑是否该引入 Redux。

了解了什么时候使用 **Redux**，那么什么时候使用 **React** 内建 **state** 呢？如果你本来就不打算引入 **Redux**，任何时候都可以使用 **React** 内建 **state**。

可否混用 **React** 内建 **state** 和 **Redux**？

我们进入今天这节课的最后一个问题：“可否混用 **React** 内建 **state** 和 **Redux**？”当然可以。当你决定了为项目引入 **Redux**，并不意味着你就与 **useState** 说再见了。它们**可以共存，而且可以配合得很好**。

一般情况下可以这样分工：

- 全局状态倾向于放到 **Redux store** 里；
- 局部状态倾向于放到 **React state** 里；
- 业务状态倾向于放到 **Redux store** 里；
- 交互状态倾向于放到 **React state** 里；
- 必要时，可以把外部状态同步到 **Redux store** 里。

这里我插一个经济学和社会学概念：**路径依赖（Path Dependence）**，说的是人们在当前做的决策选择往往受制于他过去的决策，哪怕所处环境已经发生变化。最著名的例子是 **QWERTY** 键盘的历史，感兴趣的话你可以上网搜一下，这个概念本来不具有贬义或者褒义。

我曾见过不少开发者，在 **React** 中引入 **Redux** 后，无论大大小小的状态都习惯性地往 **Redux store** 里放，这就是一种路径依赖了。这样做的后果也很容易想象得到：**Redux store 不堪重负，React 反而头重脚轻**。

小结

这节课我们继续上节课应用状态管理的内容，介绍了 **React** 应用中的三种状态：业务状态、交互状态和外部状态，并且学习了从数据流层面区分的全局状态和局部状态。

紧接着我们学习了什么时候为 **React** 项目引入 **Redux**、如何用 **React Redux** 连接器，最后总结了混用 **React** 内建 **state** 和 **Redux store** 的一些分工建议。

这节课讲了这么多应用状态，那么应用状态算不算是 **React** 应用的数据模型呢？如果是的话，该如何保证模型的 **schema**（模式）呢？下节课我们会学习如何活用 **PropTypes** 和 **TypeScript**，在 **React** 应用中加入数据类型验证。

思考题

这节课虽然写了一个很小的 **React Redux Demo**，但并不打算用 **Redux** 来改写 **oh-my-kanban** 或者 **yeah-my-kanban** 项目。


你愿意接受这个挑战吗？如果愿意的话，请建一个 **fork** 或者分支，把最新的代码用 **Redux** 改写一遍，然后在留言区留下你的代码仓库链接与大家分享，或者留一个“M”字样代表你很满意自己的成果。

你将遇到的具体挑战包括：

1. 在课程中并没有把 **Redux Toolkit** 和 **React Redux** 的所有 **API** 都展示出来，你需要参考官方文档；
2. 三个看板列状态数据结构和操作都相同，你会怎么设计 **slice**、**reducer**？当然，直接创建三个相似的 **slice** 也是完全 **ok** 的；
3. 哪些状态会放在 **Redux store** 中，哪些会放在 **React state** 中。

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 16 | 应用状态管理（上）：应用状态管理框架**Redux**

下一篇 18 | 数据类型：活用**TypeScript**做类型检查

精选留言 (1)

写留言



2022-10-08 来自北京

老师提到了路径依赖这个概念，说明老师平时的知识储备是不局限于技术领域的。学习了，向老师看齐～

作者回复: 你好，辰洋，很高兴你注意到了“路径依赖”概念。

我认为各行各业各个领域其实具有相通性，时不时地找些机会拓宽自己的视野，对自己的技术生涯和职业生涯一定会有帮助。就比如我身边一些炒股的同事，应该是从股市中学到了某些很厉害的知识技能，最近在本职工作上非常积极上进，我很佩服他们。



3