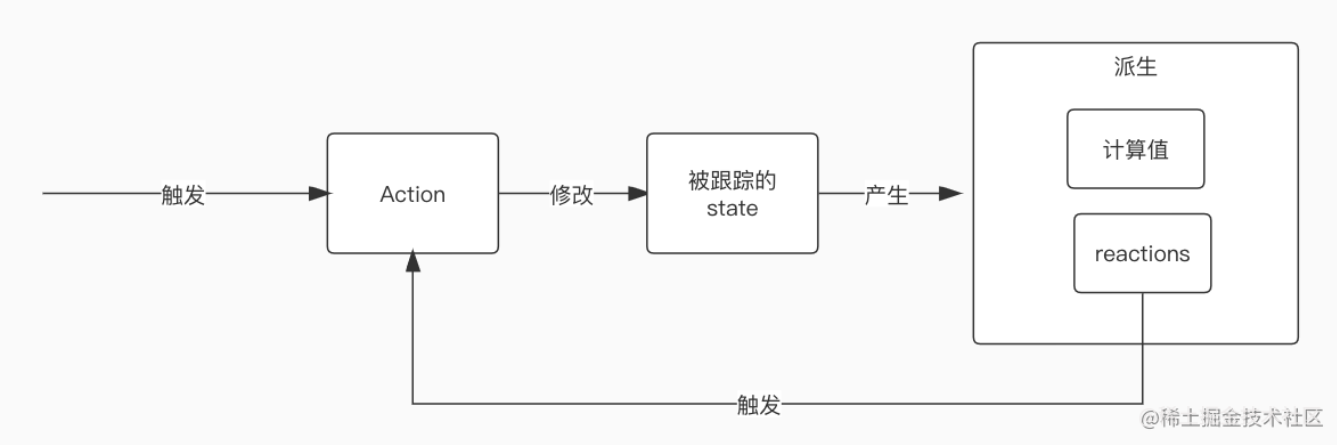
用 Mobx 实现 React 应用的状态管理

MobX 是一个状态管理库，它会自动收集并追踪依赖，开发人员不需要手动订阅状态，当状态变化之后 MobX 能够精准更新受影响的内容，另外它不要求 state 是可 JSON 序列化的，也不要求state 是 immutable，MobX 推荐的数据流如下图所示：



本文先以一个 demo 单独介绍 Mobx 的用法，再介绍如何将 Mobx 与 React 结合实现 React 应用程序的状态管理。

从一个 demo 开始

这部分用 MobX + TypeScript 实现一个 TODO List 的 demo，MobX 的版本为 6.5.0，TypeScript 的版本为 4.5.4，将 TypeScript 编译器配置项 useDefineForClassFields 设置为 true。

创建类并将其转化成可观察对象

创建 ToDoItem 类和 ToDoList 类，ToDoItem 类的代码如下：

typescript 复制代码

```typescript
import { makeObservable, observable, action } from 'mobx'
class ToDoItem {
    id: number
    name: string
    status: 0 | 1

    changeStatus(status: Status) {
```

```
        this.status = status
    }

    constructor(name: string) {
        this.id = Uid ++
        this.name = name
        this.status = 0
       // 注意这里
        makeObservable(this, {
            status: observable,
            changeStatus: action
        })
    }
}
```

用 makeObservable 将 ToDoItem 实例变成可观察的，用 observable 标记 status 字段，让 MobX 跟踪它的变化，changeStatus 方法用于修改 status 的值，所以用action标记它。

ToDoList 类比 ToDoItem 类复杂一些，它收集 Todo-List Demo 需要的全部数据，代码如下：

typescript 复制代码

```
import { makeObservable, observable, action, computed, runInAction } from 'mobx'

class ToDoList {
    searchStatus?: 0 | 1
    list: ToDoItem[] = []
    get displayList() {
        if (!this.searchStatus) {
            return this.list
        } else {
            return this.list.filter(item => item.status === this.searchStatus)
        }

    }

    changeStatus(searchStatus: Status | undefined) {
        this.searchStatus = searchStatus
    }

    addItem(name: string) {
        this.list.push(new ToDoItem(name))
    }

    async fetchInitData() {
        await waitTime()
       // 注意这里
```

```
        runInAction(() => {
            this.list = [new ToDoItem('one'), new ToDoItem('two')]
        })
    }

    constructor() {
        makeObservable(this, {
            searchStatus: observable,
            list: observable,
            displayList: computed,
            changeStatus: action,
            addItem: action
        })
    }
}
```

与 ToDoItem 相比，ToDoList 多使用了 computed 标记，这是因为 displayList 的值由 searchStatus 和 list 通过一个纯函数计算而来，所以它被标记为 computed。fetchInitData 是一个异步方法，在其中用 runInAction 创建一个立即执行的 action 去修改 list 的值，从 fetchInitData 的实现可以看出，异步修改 state 和同步修改 state 没有差别，只要保证 state 是在 action 中修改的即可。

使用可观察对象

在上一步的 ToDoList 和 ToDoItem 的构造函数中，我们调用了 makeObservable 方法，并用合适的注解去标记实例字段，接下来用一段代码验证 MobX 是否按照要求跟踪state的变化。代码如下：

typescript 复制代码

```
import { autorun} from 'mobx'

autorun(() => { console.log(toDoList.list.length) }) // line A
autorun(() => { console.log(toDoList.list) }) // line B
```

autorun 接收一个函数，该函数同步执行过程中访问的 state 或计算值发生变化时，它会自动运行，另外，调用 autorun 时，该函数也会运行一次。使用 toDoList.addItem 方法往 list 数组中 push 一个事项，你会发现上述 line A 的函数会运行，但是 line B 的函数不会运行；使用 toDoList.fetchInitData 方法给 list 数组赋值，line A 和 line B 的函数都会运行，出现这种差异是因为 autorun 使用全等（===）运算符确定两个值是否相等，但它认为 NaN 等于 NaN 。

用如下一段代码验证 MobX 是否按照要求跟踪 ToDoItem 实例的 state 的变化：

```typescript 复制代码
import { autorun} from 'mobx'
autorun(() => {
  if (toDoList.list.length) {
       console.log(toDoList.list[0]?.status)
   }
})
reaction(() => toDoList.list.length, () => {
  toDoList.list[0].changeStatus(1)// 修改status的值
})
```

当 reaction 的第一个参数返回 true 时，它的第二个参数会自动执行,上述代码在 reaction 中修改 toDoItem 的 status 字段，修改之后 autorun 能成功运行一次。

MobX 与 React 集成

现在将上一步的 TODO List 与 React 结合，为此需要安装 mobx-react-lite 或 mobx-react，mobx-react 比 mobx-react-lite 的功能更多，同时它的体积也更大，如果你的项目只使用函数组件，那么推荐安装 mobx-react-lite 而非 mobx-react，为了演示更多的用法本小节安装 mobx-react。另外，本小节会用到装饰器语法，所以要将 TypeScript 编译器配置项 experimentalDecorators 设置为true。下面是一个 MobX + React 的简单示例：

```tsx 复制代码
import { observer } from 'mobx-react'
import toDoList, { Status } from '../../mobx/todo'

const ToDoListDemoGlobalInstance= observer(
  class extends React.Component<{}, {}> {
    componentDidMount() {
     // 3s之后修改 searchStatus 的值
      setTimeout(() => {
        toDoList.changeStatus(Status.finished)
      }, 3000);
    }

    render() {
      return (
        <div>searchStatus: {toDoList.searchStatus}</div>
      )
    }
  }
)
```

observer 是一个高阶组件，它会订阅组件在渲染期间访问的可观察对象，可观察对象指的是用 makeAutoObservable 、makeObservable 或 observable 转换之后的对象，当组件渲染期间访问的 state 和计算值发生变化时，组件会重新渲染。上述代码，组件被装载 3s 后将修改 searchStatus 的值，由于 render 方法访问了 searchStatus 的值，所以组件会重新渲染。observer 除了以高阶组件的形式使用之外，还能以装饰器的形式使用。

在组件中使用可观察对象

下面介绍 6 种在组件中使用 MobX 可观察对象的写法。

1. 访问全局的类实例

上一个示例代码便是在组件中直接访问全局的类实例，在这里不再举更多的示例代码。

2. 通过 props

这种方式是指将可观察对象通过 props 的形式传递到组件中，代码如下：

```typescript 复制代码
import { observer } from 'mobx-react'
import toDoList, { Status } from '../../mobx/todo'

@observer
class ToDoListDemoByProps extends React.Component<{toDoList: ToDoList}, {}> {
  componentDidMount() {
    setTimeout(() => {
      toDoList.changeStatus(Status.finished)
    }, 3000);
  }

  render() {
   // 读取props中的可观察对象
    return (
      <div>ToDoListDemoByProps - searchStatus: {this.props.toDoList.searchStatus}</div>
    )
  }
}

//使用ToDoListDemoByProps
<ToDoListDemoByProps toDoList={toDoList}/>
```

3. 通过 React Context

这种方式是指通过 React Context 让可观察对象在整个被 Context.Provider 包裹的组件树中共享，代码如下：

```tsx 复制代码
import { observer } from 'mobx-react'
import toDoList, { Status, ToDoList } from '../../mobx/todo'

// 创建一个用observer包裹的函数组件
const ToDoListDemoByContext = observer(() => {
 // 在函数组件中使用Context
  const context = useContext(todoContext);
  useEffect(() => {
    setTimeout(() => {
      context.changeStatus(Status.finished)
    }, 3000);
  })
  return (
    <div>ToDoListDemoByContext - searchStatus: {context.searchStatus}</div>
  )
})

// 往Context传值
<todoContext.Provider value={toDoList}>
   <ToDoListDemoByContext/>
</todoContext.Provider>
```

4. 在组件中实例化 observable class 并存储它的实例

这种方式指的是在组件作用域中实例化类，并且将结果保存到组件的某个字段中，如果在函数组件中使用这种方式，那么还需要用到 useState。代码如下：

```typescript 复制代码
const ToDoListFuncDemoLocalInstance= observer(() => {
  // 实例化类
  const [ todoList ] = useState(() => new ToDoList())
  useEffect(() => {
    setTimeout(() => {
      // 使用实例方法更新状态
      todoList.changeStatus(Status.finished)
    }, 3000);
  })
  return (
    <div>ToDoListDemoLocalInstance - searchStatus: {todoList.searchStatus}</div>
  )
})
```

对于类组件而言，只需要将 new ToDoList() 的结果保存在它的实例属性上，之后在组件中访问该实例属性，代码如下：

typescript 复制代码

```typescript
@observer
class ToDoListClassDemoLocalInstance extends React.Component<{}, {}> {
  todoList = new ToDoList()
  // other
}
```

5. 在组件中调用 observable 方法创建可观察对象

这种方式不使用类去创建可观察对象，而是使用 observable 方法创建可观察对象，与第 4 种方式一样，如果在函数组件中还要用到 useState，代码如下

javascript 复制代码

```javascript
import { observable } from 'mobx'

const LocalObservableDemo = observer(() => {
 // 调用observable
  const [counter] = useState(() => observable({
    count: 0,
    addCount() {
      this.count ++
    }
  }))

  return <>
    <div>{counter.count}</div>
    <button onClick={() => counter.addCount()}>add</button>
  </>
})
```

上述代码使用 mobx 导出的 observable 方法创建一个可观察对象，并在函数组件使用该对象，当它的 count 属性值发生变化时，组件将重新渲染。对于类组件而言只需要将 observable 函数的结果保存到实例属性上即可。

6. 在函数组件中使用 useLocalObservable

useLocalObservable 是 useState + observable 简写版本，只能在函数组件中使用，代码如下：

```tsx 复制代码
import { observer, useLocalObservable } from 'mobx-react'
const UseLocalObservableDemo = observer(() => {
  const counter = useLocalObservable(() => ({
    count: 0,
    addCount() {
      this.count ++
    }
  }))

  return <>
    <div>{counter.count}</div>
    <button onClick={() => counter.addCount()}>add</button>
  </>
})
```

对于函数组件而言，useLocalObservable 只是一个自定义Hook，它返回一个可观察对象。

有多种方式让组件在渲染阶段使用可观察对象，不管是哪种方式，组件都必须具备观察能力，否则，当渲染期间访问的 state 和计算值发生变化时，组件不会重新渲染。笔者在使用 MobX 做状态管理时，最常用的方式是第 1 和第 2 种。第 4、5、6 种方式必要性不大。

让组件具备观察能力

observer 是让组件具备观察能力最常见的方式，在这里介绍另一种让组件具备观察能力的方式，即：Observer组件。用法如下：

```typescript 复制代码
import { Observer } from 'mobx-react'

class ObservableDemo extends React.Component<{},{}> {
  render() {
    return (
    <>
      <div>{toDoList.searchStatus || '-'}</div>  {/** lineA */}
      <Observer>
        {() => <div>{toDoList.searchStatus || '-'}</div>} {/** lineB */}
        </Observer>
    </>
    )
  }
}
```

Observer 组件会创建一个匿名的观察区域，在上述代码中，如果 toDoList.searchStatus 的值发生变化，那么 lineB 会重新渲染，但是 lineA 不会重新渲染。

总结

将 MobX 与 React 结合在一起的关键在于用 observer 包裹组件以及在组件中读取可观察对象，observer 不关心可观察对象从哪里来，也不关心如何读取可观察对象，只关心在组件中可观察对象是否可读。对于习惯面向对象编程的工程师而言，用 MobX 做状态管理会比用 Redux 做状态管理更得心应手。