react中的副作用?

开篇

最近在研究 `react源码` 嘛，当正好我研究到 `hooks` 的时候，脑海就忍不住想了有关 `react副作用` 的连续问题，这这篇文章我们只少量的分析源码，主要是描述一个思考的思路。先玩点抽象的概念，`react` 就像个女孩，她不是很纯洁 `effect`，但她很尽力的表现得她很 `纯洁`，当我们去使用 `useEffect` 可以从中窥得她的一部分 `不纯洁`。

思考过程

什么是副作用 -> React中的副作用，以及怎么做副作用分离 -> 简单的看看实现

副作用

先想想我们平常如果在 `日常` 会怎么理解 `副作用`，这个词经常会用在我们日常生活中的 `吃药`，`比如:` 我原本要吃一片感冒药为了治好我的 `感冒`，但是这药，在 `治好的感冒` 的同时可能还造成了 `我的拉肚子`。

我们可以把这个例子反馈到我们的函数中，`治好感冒` 是我们的目的是我们 `函数返回值`，而 `拉肚子` 是在 `我们获得这个函数返回值` 的过程中的 `副作用`，这个描述可能不太准确，比较准确的说法 `是函数在执行过程中对外部造成的影响称之为副作用`。

我们可以举一个最简单的例子：下述代码，除了返回了我们的 `6`，还同时产生了 `附加的影响`，将 `arr` 数组删除了最后一项。而这个 `附加的影响` 就可以被称之为副作用。（有关 `纯函数，非纯函数，引用透明` 这些概念性编程范式的东西，可以去看看其他博主的，网上有挺多的。）

```js
let arr = [1,2,3,4,5,6];
let lastOne = arr.pop();
```

React副作用分离

这里就涉及一个知识点，代数效应（算了这东西有很多解释大家可以去看看，强抓八股文并不利于我们思考）。简单的说就是在 `React hook` 中以这个理念，去实现了一套让我们在 `api` 使用层面可以去不关注 `副作用` ，将 `副作用` 分离出来，交给内部处理，举个例子： `本质上useState是一个有副作用的api但是react内部帮你抽离了让你在使用层面上看起来像一个纯函数` 。

下述例子， `useEffect` 函数中我们可以不用去关心可能它在 `count` 改变后，对 `fiber` 渲染所做的事情，他在内部已经给你处理了。其实到这不关心源码的朋友们已经可以结束了。

```js
const [count, setCount] = useState(0)
useEffect(() => {
  // 随便做点啥
}, [count])
```

是如何分离的，又拿副作用做了什么?

因为实际上这个问题太大了， `react` 的内部处处是副作用，但我们可以单独从一个小点 `useEffect` 去分析，因为我们翻译一下就可以知道这个 `hook` 就是抛出的最直观的去 `使用副作用` 的地方， `useEffect` 的调用时机是在 `fiber` 树渲染的过程中，他可以改变状态，发起新的一轮的 `fiber` 构造等去引发一个连续的 `副作用` ，其实本质上感觉 `hook` 或者说后续回调函数完成后的 `修改了状态，对函数以外的东西产生了影响` 都可以视为 `副作用` 。

简单说说把， `useEffect` 会创建一个 `effect` 环形链表并保存在 `fiber.updateQueue` 和 `hook.memoizedState` 中，当我们渲染开始的时候会将 `useEffect` 放入调度中，等到 `flushPassiveEffectsImpl` 去异步执行它。此时会有一些副作用留存到 `fiber` 上。

```js
try {
   const create = effect.create;
   if (
     enableProfilerTimer &&
     enableProfilerCommitHooks &&
     fiber.mode & ProfileMode
   ) {
     try {
       startPassiveEffectTimer();
```

```
        effect.destroy = create();
      } finally {
        recordPassiveEffectDuration(fiber);
      }
    } else {
      // 执行
      effect.destroy = create();
    }
    } catch (error) {
    invariant(fiber !== null, 'Should be working on an effect.');
    captureCommitPhaseError(fiber, error);
    }
```

等到我们更新的时候，这些副作用又会派上用场，依次循坏。

```
function updateEffectImpl(fiberFlags, hookFlags, create, deps): void {
  // 获取当前hook
  const hook = updateWorkInProgressHook();
  const nextDeps = deps === undefined ? null : deps;
  let destroy = undefined;
  // 分析依赖
  if (currentHook !== null) {
    // 留存下来的`effect`链表
    const prevEffect = currentHook.memoizedState;
    // 继续使用先前effect.destroy
    destroy = prevEffect.destroy;
    if (nextDeps !== null) {
      const prevDeps = prevEffect.deps;
      // 浅比较依赖是否变化
      if (areHookInputsEqual(nextDeps, prevDeps)) {
        如果依赖不变，新建effect(tag不含HookHasEffect)
        pushEffect(hookFlags, create, destroy, nextDeps);
        return;
      }
    }
  }
  // 如果依赖改变，更改fiber.flag，新建effect
  currentlyRenderingFiber.flags |= fiberFlags;

  hook.memoizedState = pushEffect(
    HookHasEffect | hookFlags,
    create,
    destroy,
    nextDeps,
  );
}
```