从React源码分析看useEffect

热身准备

这里不再讲 `useLayoutEffect`，它和 `useEffect` 的代码是一样的，区别主要是：

- 执行时机不同；
- `useEffect` 是异步，`useLayoutEffect` 是同步，会阻塞渲染；

初始化 mount

mountEffect

在所有 `hook` 初始化时都会通过下面这行代码实现 `hook` 结构的初始化和存储，这里不再讲 `mountWorkInProgressHook` 方法

```javascript 复制代码
var hook = mountWorkInProgressHook();
```

在 `mountEffect` 方法中，只有这几行代码。先来解读下几个参数：

- fiberFlags：有副作用的更新标记，用来标记hook所在的 `fiber`；
- hookFlags：副作用标记；
- create：使用者传入的回调函数；
- deps：使用者传入的数组依赖；

```javascript 复制代码
function mountEffectImpl(fiberFlags, hookFlags, create, deps) {
  // hook初始化
  var hook = mountWorkInProgressHook();
  // 判断是否有传入deps，如果有会作为下次更新的deps
  var nextDeps = deps === undefined ? null : deps;
  // 给hook所在的fiber打上有副作用的更新的标记
  currentlyRenderingFiber$1.flags |= fiberFlags;
  // 将副作用操作存放到fiber.memoizedState.hook.memoizedState中
  hook.memoizedState = pushEffect(HasEffect | hookFlags, create, undefined, nextDeps);
}
```

上面代码中都有注释，接下来我们看看 `React` 是如何存放副作用更新操作的，主要就是 `pushEffect` 方法

```javascript
function pushEffect(tag, create, destroy, deps) {
  // 初始化副作用结构，
  var effect = {
    tag: tag,
    create: create,   // 回调函数
    destroy: destroy,  // 回调函数里的return（mount时是undefined）
    deps: deps,    // 依赖数组
    // 闭环链表
    next: null
  };
  // 下面的一大段代码看着复杂，但是有没有很熟悉的感觉？
  var componentUpdateQueue = currentlyRenderingFiber$1.updateQueue;

  if (componentUpdateQueue === null) {
    componentUpdateQueue = createFunctionComponentUpdateQueue();
    currentlyRenderingFiber$1.updateQueue = componentUpdateQueue;
    // effect.next = effect形成环形链表
    componentUpdateQueue.lastEffect = effect.next = effect;
  } else {
    var lastEffect = componentUpdateQueue.lastEffect;

    if (lastEffect === null) {
      componentUpdateQueue.lastEffect = effect.next = effect;
    } else {
      var firstEffect = lastEffect.next;
      lastEffect.next = effect;
      effect.next = firstEffect;
      componentUpdateQueue.lastEffect = effect;
    }
  }

  return effect;
}
```

上面这段代码除了初始化副作用的结构代码外，都是我们前面讲过的操作闭环链表，向链表末尾添加新的 `effect` ，该 `effect.next` 指向 `fisrtEffect` ，并且链表当前的指针指向最新添加的 `effect` 。

`useEffect` 的初始化就这么简单，简单总结一下：给 `hook` 所在的 `fiber` 打上副作用更新标记，并且 `fiber.memoizedState.hook.memoizedState` 和 `fiber.updateQueue` 存储了相关的副作用，这些副作用通过闭环链表的结构存储。

更新 update

updateEffect

`updateWorkInProgressHook` 在上篇文章也已讲过，不再详述，主要功能就是创建一个带有回调函数的 `newHook` 去覆盖之前的 `hook` 。

```javascript
function updateEffectImpl(fiberFlags, hookFlags, create, deps) {
  var hook = updateWorkInProgressHook();
  var nextDeps = deps === undefined ? null : deps;
  var destroy = undefined;

  if (currentHook !== null) {
    var prevEffect = currentHook.memoizedState;
    destroy = prevEffect.destroy;

    if (nextDeps !== null) {
      var prevDeps = prevEffect.deps;
      // 比较两次依赖数组中的值是否有变化
      if (areHookInputsEqual(nextDeps, prevDeps)) {
        // 和之前初始化时一样
        pushEffect(hookFlags, create, destroy, nextDeps);
        return;
      }
    }
  }
  // 和之前初始化时一样
  currentlyRenderingFiber$1.flags |= fiberFlags;
  hook.memoizedState = pushEffect(HasEffect | hookFlags, create, destroy, nextDeps);
}
```

相信眼眼尖的看官已经注意到上面代码中有两个 `pushEffect` ，一个没有赋值给 `hook.memoizedState` ，一个赋值了，这两者有什么区别呢？

先保留着这个疑问，先来了解下下面这行代码都做了些什么，因为它造就了两个 `pushEffect` 。

`if (areHookInputsEqual(nextDeps, prevDeps)){...}`

```kotlin
function areHookInputsEqual(nextDeps, prevDeps) {
  // 没有传deps的情况返回false
  if (prevDeps === null) {
    return false;
```

```
  }
  // deps不是[]，且其中的值有变动才会返回false
  for (var i = 0; i < prevDeps.length && i < nextDeps.length; i++) {
    if (objectIs(nextDeps[i], prevDeps[i])) {
      continue;
    }
    return false;
  }
  // deps = []，或者deps里面的值没有变化会返回true
  return true;
}
```

它会判断两次依赖数组中的值是否有变化以及 `deps` 是否是空数组来决定返回 `true` 和 `false`，返回 `true` 表明这次不需要调用回调函数。

现在我们明白了两次 `pushEffect` 的异同， `if` 内部的 `pushEffect` 是不需要调用的回调函数，外面的 `pushEffect` 是需要调用的。再来仔细看下这两行代码：

javascript 复制代码

```
// if内部的，第一个参数是hookFlags = 4
pushEffect(hookFlags, create, destroy, nextDeps);
// if外部的，第一个参数是HasEffect | hookFlags = 5
hook.memoizedState = pushEffect(HasEffect | hookFlags, create, destroy, nextDeps);
```

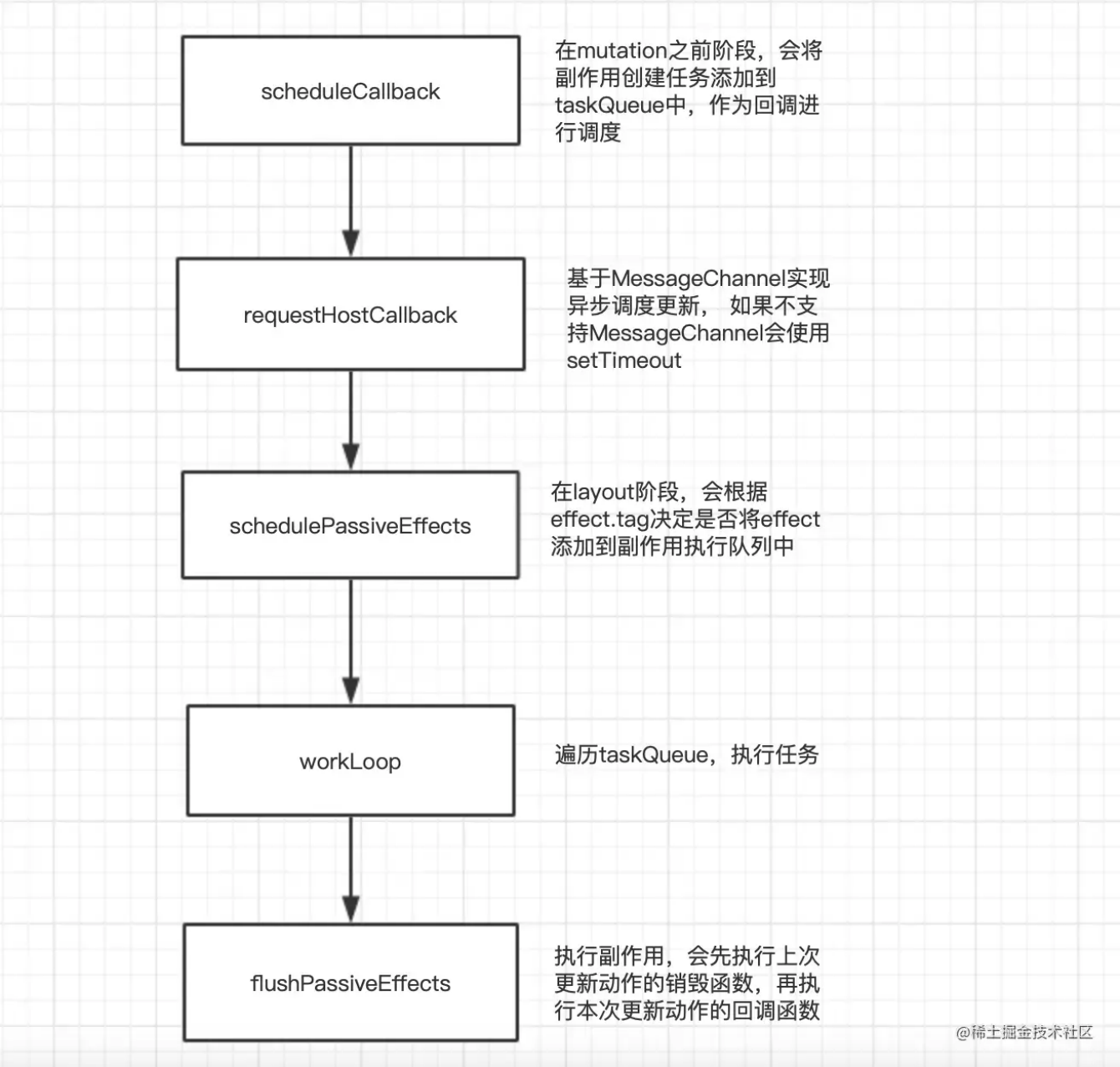
相关参考视频讲解：进入学习

这两行代码的区别是传入的第一个参数不同，而第一个参数就是 `effect.tag` 的值， `effect.tag = 4` 不会添加到副作用执行队列，而 `effect.tag = 5` 可以。没有添加到副作用执行队列的 `effect` 就不会执行。这样就巧妙的实现了 `useEffect` 基于 `deps` 来判断是否需要执行回调函数。

到这里， 我们搞明白了，不管 `useEffect` 里的 `deps` 有没有变化都会为回调函数创建 `effect` 并添加到 `effect` 链表和 `fiber.updateQueue` 中，但是 `React` 会根据 `effect.tag` 来决定该 `effect` 是否要添加到副作用执行队列中去执行。

执行副作用

我们现在知道了， `useEffect` 是异步执行的。那么这个回调函数副作用会在什么时候执行呢？ `useEffect` 回调函数会在 `layout` 阶段之后执行。现在我们来了解下具体调用执行的流程。



我画了一个简单的流程图，大致描述了下调用流程。首先在 `mutation` 之前阶段，基于副作用创建任务并放到 `taskQueue` 中，同时会执行 `requestHostCallback` ，这个方法就涉及到了异步了，它首先考虑使用 `MessageChannel` 实现异步，其次会考虑使用 `setTimeout` 实现。使用 `MessageChannel` 时， `requestHostCallback` 会马上执行 `port.postMessage(null);` ，这样就可以在异步的第一时间执行 `workLoop` ， `workLoop` 会遍历 `taskQueue` ，执行任务，如果是 `useEffect` 的 `effect` 任务，会调用 `flusnPassiveEffects` 。

Q：可能有人会疑惑为什么优先考虑 `MessageChannel` ？

A： 首先我们要明白 `React` 调度更新的目的是为了时间分片，意思是每隔一段时间就把主线程还给浏览器，避免长时间占用主线程导致页面卡顿。使用 `MessageChannel` 和 `SetTimeout` 的目的都是为了创建**宏任务**，因为**宏任务**会在当前微任务都执行完后，等到浏览器主线程空闲后才

会执行。不优先考虑 `setTimeout` 的原因是， `setTimeout` 执行时间不准确，会造成时间浪费，即使是 `setTimeout(fn, 0)` ，感兴趣的可以去自己了解下，本文不做赘述了。

在 `schedulePassiveEffects` 中，会决定是否执行 `effect` 链表中的 `effect` ，判断的依据就是每个 `effect` 上的 `effect.tag` ：

javascript 复制代码

```
function schedulePassiveEffects(finishedWork) {
  var updateQueue = finishedWork.updateQueue;
  var lastEffect = updateQueue !== null ? updateQueue.lastEffect : null;

  if (lastEffect !== null) {
    var firstEffect = lastEffect.next;
    var effect = firstEffect;
    // 遍历effect链表
    do {
      var _effect = effect,
          next = _effect.next,
          tag = _effect.tag;
      // 基于effect.tag决定是否添加到副作用执行队列
      if ((tag & Passive$1) !== NoFlags$1 && (tag & HasEffect) !== NoFlags$1) {
        enqueuePendingPassiveHookEffectUnmount(finishedWork, effect);
        enqueuePendingPassiveHookEffectMount(finishedWork, effect);
      }

      effect = next;
    } while (effect !== firstEffect);
  }
}
```

在 `flushPassiveEffects` 中，会先执行上次更新动作的销毁函数，然后再执行本次更新动作的回调函数，并且会把回调函数的 `return` 作为下次更新动作的销毁函数。

javascript 复制代码

```
function flushPassiveEffectsImpl() {
  // 执行上次更新动作的销毁函数
  var unmountEffects = pendingPassiveHookEffectsUnmount;
  pendingPassiveHookEffectsUnmount = [];
  for (var i = 0; i < unmountEffects.length; i += 2) {
    ...destroy()
  }
  // 执行本次更新动作的回调函数
  var mountEffects = pendingPassiveHookEffectsMount;
  pendingPassiveHookEffectsMount = [];
  for (var _i = 0; _i < mountEffects.length; _i += 2) {
    ...create()
```

```
  }
}
```

上面代码中的这两行就是来自副作用执行队列，已经过滤掉了不需要执行的 `effect`，只执行该队列上的副作用函数

```javascript
var unmountEffects = pendingPassiveHookEffectsUnmount;

var mountEffects = pendingPassiveHookEffectsMount;
```

总结

看完这篇文章， 我们可以弄明白下面这几个问题：

1. `useEffect` 和 `useLayoutEffect` 的区别?
2. `useEffect` 是怎么判断回调函数是否需要执行的?
3. `useEffect` 是同步还是异步?
4. `useEffect` 是通过什么实现异步的?
5. `useEffect` 为什么要要优先选用 `MessageChannel` 实现异步?