【React学习】React更新渲染原理

当我们调用 setState 之后发生了什么？react经历了怎样的过程将新的 state 渲染到页面上？

一次 `react` 更新，核心就是对虚拟 `dom` 进行 `diff`，找出最少的需要变化的 `dom` 节点，然后对其进行相应的 `dom` 操作，用户即可在页面上看到更新。但 react 作为广泛使用的框架，需要考虑更多的因素，考虑多个更新的优先级，考虑主线程占用时长，考虑 `diff` 算法复杂度，考虑性能。。等等，本文就来探讨一下react在其内部是如何处理数据更新的。

react在内部使用 `fiber` 这种数据结构来作为虚拟dom【react16+】，它与 `dom tree` 一一对应，形成 `fiber tree`，一次react更新，本质是 `fiber tree` 结构的更新变化。而 `fiber tree` 结构的更新，用更专业的术语来讲，其实就是 `fiber tree` 的协调（Reconcile）。 `Reconcile` 中文意思是调和、使一致，协调fiber tree，就是调整 `fiber tree` 的结构，使其和更新后的 `jsx` 模版结构、 `dom tree` 保持一致。

react从16起，将更新机制分为三个模块，也可以说是三个步骤，分别是 `Schedule` 【调度】、 `Reconcile` 【协调】、 `render` 【渲染】

Schedule

为什么需要Schedule?

首先我们要知道react在进行协调时，提供了两种模式： `Legacy mode` 同步阻塞模式和 `Concurrent mode` 并行模式。

不同上下文中的更新会触发不同的模式，如果是在 `event` 、 `setTimeout` 、 `network request` 的 `callback` 中触发更新，react 会采用 `Legacy` 模式。如果更新与 `Suspense` 、 `useTransition` 、 `OffScreen` 相关，那么 react 会采用 `Concurrent` 模式。

Legacy mode

`Legacy mode` 在协调时会启动 `workLoopSync` 。 `workLoopSync` 开始工作以后，要等到所有 `fiber node` 都处理完毕以后，才会结束工作，也就是 `fiber tree` 的协调过程不可中断。

`Legacy mode` 存在的问题：如果 `fiber tree` 的结构很复杂，那么协调 `fiber tree` 可能会占用大量的时间，导致主线程会一直被 js 引擎占用，渲染引擎无法在规定时间(浏览器刷新频率 - 16.7ms)内完成工作，使得页面出现卡顿(掉帧)，影响用户体验。

Concurrent mode

鉴于 `Legacy mode` 存在的问题，react团队在 `react 16` 中提出了 `Concurrent mode` 的概念，并在 `react 18` 中开放使用。react16、17一直为此做准备。

`Concurrent` 模式最大的意义在于，使用 `Concurrent` 模式以后的react的应用可以做到：

- 协调可以中断、恢复；不会长时间阻塞浏览器渲染
- 高优先级更新可以中断低优先级更新，优先渲染

那么，怎么做到这两点呢?

事实上，`Schedule` 就是用来完成这个任务的，调度任务的优先级，使高优先级任务优先进入 `Reconcile`，并且提供中断和恢复机制。

时间切片

`react` 采用时间切片的方式来实现协调的中断和恢复，`Concurrent mode` 在协调时会启动 `workLoopConcurrent`。 `workLoopConcurrent` 开始工作以后，每次协调 `fiber node` 时，都会判断当前时间片是否到期。如果时间片到期，会停止当前 `workLoopConcurrent`，让出主线程，然后请求下一个时间片继续协调。

协调的中断及恢复，类似于浏览器的 `eventloop`，js引擎和渲染引擎互斥，在主线程中交替工作。

我们可以通过模拟 `eventLoop` 来实现时间分片以及重新请求时间片。一段 js 程序，如果在规定时间内没有结束，那我们可以主动结束它，然后请求一个新的时间片，在下一个时间片内继续处理上一次没有结束的任务。

```scss
let taskQueue = [];   // 任务列表
let shouldTimeEnd = 5ms;   // 一个时间片定义为 5ms
let channel = new MessageChannel();  // 创建一个 MessageChannel 实例

function wookLoop() {
```

```
    let beginTime = performance.now();  // 记录开始时间
    while(true) { // 循环处理 taskQueue 中的任务
        let currentTime = performance.now();  // 记录下一个任务开始时的时间
        if (currentTime - beginTime >= shouldTimeEnd) break;  // 时间片已经到期，结束任务处理
        processTask();  // 时间片没有到期，继续处理任务
    }
    if (taskQueue.length) { // 时间片到期，通过调用 postMessage，请求下一个时间片
        channel.port2.postMessage(null);
    }
}

channel.port1.onmessage = wookLoop;  // 在下一个时间片内继续处理任务
workLoop();
```

和浏览器的消息队列 一样， `react` 也会维护一个任务队列 `taskQueue`，然后通过 `workLoop` 遍历 `taskQueue`，依次处理 `taskQueue` 中的任务。

`taskQueue` 中收集任务是有先后处理顺序的，`workLoop` 每次处理 `taskQueue` 中的任务时，都会挑选优先级最高的任务进行处理。

每触发一次 `react` 更新，意味着一次 `fiber tree` 的协调，但协调并不会在更新触发时立刻同步进行。相反，react 会为这一次更新，生成一个 `task`，并添加到 `taskQueue` 中，`fiber tree` 的协调方法会作为新建 `task` 的 `callback`。当 `wookLoop` 开始处理该 `task` 时，才会触发 `task` 的 `callback`，开始 `fiber tree` 的协调。

任务的优先级

react在内部定义了 5 种类型的优先级，以及对应的超时时间 `timeout`

- `ImmediatePriority`，直接优先级，对应用户的 `click`、`input`、`focus` 等操作；`timeout` 为 -1，表示任务要尽快处理;
- `UserBlockingPriority`，用户阻塞优先级，对应用户的 `mousemove`、`scroll` 等操作；`timeout` 为 250 ms;
- `NormalPriority`，普通优先级，对应网络请求、`useTransition` 等操作；`timeout` 为 5000 ms;
- `LowPriority`，低优先级(未找到应用场景)；`timeout` 为 10000 ms;
- `IdlePriority`，空闲优先级，如 `OffScreen`；`timeout` 为 1073741823 ms;

5 种优先级的顺序为: `ImmediatePriority` > `UserBlockingPriority` > `NormalPriority` > `LowPriority` > `IdlePriority`。

在确定了任务的优先级以后，react 会根据优先级为任务计算一个过期时间 `expirationTime`，即 `expirationTime = currentTime + timeout`，然后根据 `expirationTime` 时间来决定任务处理的先后顺序。

`expirationTime` 越小的任务会被排在 `task` 队列的越前面，之所以需要 `timeout`，而不是直接对比优先级等级，是为了避免低优先级任务长时间被插队而导致一直无响应；同时，在时间分片到期时，需要根据 `expirationTime` 判断下一个要处理的任务是否过期，如果已过期，就不能让出主线程，需要立即处理。

⚠注：react17中用Lanes重构了优先级算法，此处不展开陈述，有兴趣的同学可查阅相关文档。

获取最先处理的task

react 采用了 `小顶堆` 来存储 `task`，实现最小优先队列，即 `taskQueue` 是一个小顶堆，放在堆顶的 `task` 是需要最先处理的。

使用最小堆时，有三个操作：`push`、`pop`、`peek`。

- `push`，入堆操作，即将 `task` 添加到 `taskQueue` 中。添加一个新创建的 `task` 时，会将 `task` 添加到最小堆的堆底，然后对最小堆做自底向上的调整。调整时，会比较堆节点（`task`）的 `expirationTime`，将 `expirationTime` 较小的 `task` 向上调整。

- `peek`，获取堆顶元素，即获取需要最先处理的 `task`，执行 `task` 的 `callback`，开始 `fiber tree` 的协调。

- `pop`，堆顶元素出堆，即 `task` 处理完毕，从 `taskQueue` 中移除。移除堆顶元素以后，会对最小堆做自顶向下的调整。调整时，也是比较堆节点(`task`) 的 `expirationTime`，将 `expirationTime` 较大的 `task` 向下调整。

高优先级的更新中断低优先级的更新

`Concurrent` 模式下，如果在低优先级更新的协调过程中，有高优先级更新进来，那么高优先级更新会中断低优先级更新的协调过程。

每次拿到新的时间片以后，`workLoopConcurrent` 都会判断本次协调对应的优先级和上一次时间片到期中断的协调的优先级是否一样。如果一样，说明没有更高优先级的更新产生，可以继续上次未完成的协调；如果不一样，说明有更高优先级的更新进来，此时要清空之前已开始的协

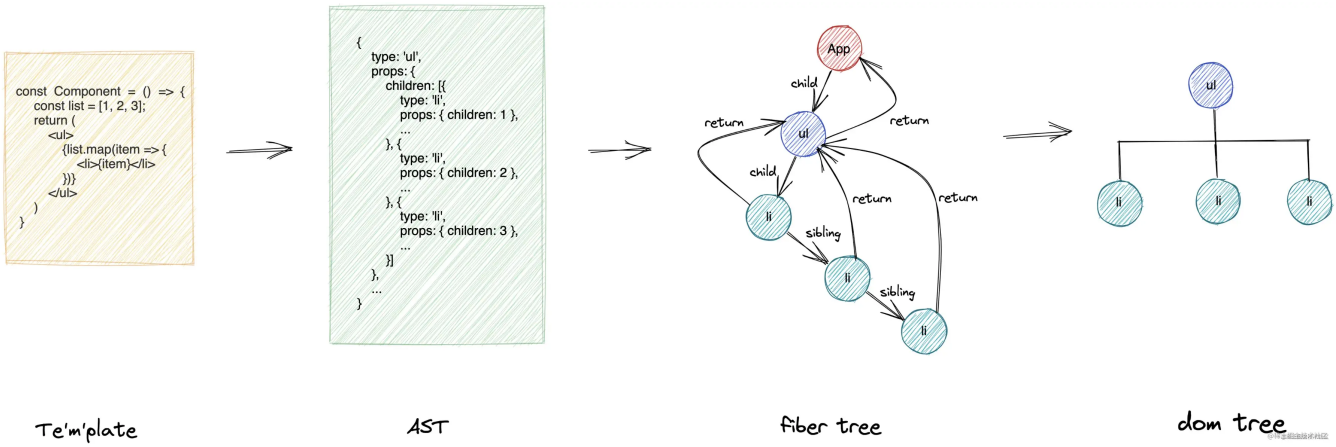
调过程，从根节点开始重新协调。等高优先级更新处理完成以后，再次从根节点开始处理低优先级更新。

Reconcile

前面说到，`reconcile`（协调）就是`fiber tree`结构的更新，那么具体是怎样更新的呢？本小节就来解答这个问题。

前置知识

从jsx到dom



Step1: 从`jsx`生成`react element`：

`jsx`模板通过`babel`编译为`createElement`方法；
执行组件方法，触发`createElement`的执行，返回`react element`；

Step2: 从`react element`生成`fiber tree`：

- `fiber tree`中存在三种类型的指针`child`、`sibling`、`return`。其中，`child`指向第一个子节点，`sibling`指向兄弟节点，`return`指针指向父节点；
- `fiber tree`采用的深度优先遍历，如果节点有子节点，先遍历子节点；子节点遍历结束以后，再遍历兄弟节点；没有子节点、兄弟节点，就返回父节点，遍历父节点的兄弟节点；
- 当节点的`return`指针返回`null`时，`fiber tree`的遍历结束；

Step3: `fiber tree` 生成之后，从 `fiber tree` 到真实 `dom`，就是处理 `fiber tree` 上对应的副作用，包括：

- 所有 `dom` 节点的新增；
- `componentDidMount`、`useEffect` 的 `callback` 函数的触发；
- `ref` 引用的初始化；

双缓存fiber tree

react 做更新处理时，会同时存在两颗 `fiber tree`。一颗是已经存在的 `old fiber tree`，对应当前屏幕显示的内容，称为 `current fiber tree`；另外一颗是更新过程中构建的 `new fiber tree`，称为 `workInProgress fiber tree`。

`current fiber tree` 和 `workInProgress fiber tree` 可以通过 `alternate` 指针互相访问

当更新完成以后，使用 `workInProgress fiber tree` 替换掉 `current fiber tree`，作为下一次更新的 `current fiber tree`。

协调的过程

协调过程中主要做三件事情：

1.为 `workInProgress fiber tree` 生成 `fiber node`；

2.为发生变化的 `fiber node` 标记副作用 `effect`；

3.收集带 `effect` 的 `fiber node`；

生成workInProgress fiber tree

`workInProgress fiber tree` 作为一颗新树，生成 `fiber node` 的方式有三种：

- 克隆(浅拷贝) `current fiber node`，意味着原来的 `dom` 节点可以复用，只需要更新 `dom` 节点的属性，或者移动 `dom` 节点；
- 新建一个 `fiber node`，意味着需要新增加一个 `dom` 节点；
- 直接复用 `current fiber node`，表示对应的 `dom` 节点完全不用做任何处理；

复用的场景：当子组件的渲染方法(类组件的 `render` 、函数组件方法)没有触发，（比如使用了 `React.memo` ），没有返回新的 `react element` ，子节点就可以直接复用 `current fiber node` 。

在日常开发过程中，我们可以通过合理使用 `ShouldComponentUpdate` 、 `React.memo` ，阻止不必要的组件重新 `render` ，通过直接复用 `current fiber node` ，加快 `workInProgress fiber tree` 的协调，达到优化的目的。

相反，只要组件的渲染方法被触发，返回新的 `react element` ，那么就需要根据新的 `react element` 为子节点创建 `fiber node` （通过浅拷贝或新建）。

- 如果能在 `current fiber tree` 中找到匹配节点，那么可以通过克隆(浅拷贝) `current fiber node` 的方式来创建新的节点；
- 相反，如果无法在 `current fiber tree` 找到匹配节点，那么就需要重新创建一个新的节点；

我们常说的 `diff算法` 就是发生在这一环节。

`diff算法` 比较的双方是 `workInProgress fiber tree` 中用于构建 `fiber node` 的 `react element` 和 `current fiber tree` 中的 `fiber node` ，比较两者的 `key` 和 `type` ，根据比较结果来决定如何为 `workInProgress fiber tree` 创建 `fiber node` 。

【 key 和 type 】：

`key` 就是 `jsx` 模板中元素上的 `key` 属性。如果不写默认为 `undefined` 。 `jsx` 模板转化为 `react element` 后，元素的 `key` 属性会作为 `react element` 的 `key` 属性。同样的， `react element` 转化为 `fiber node` 以后， `react element` 的 `key` 属性也会作为 `fiber node` 的 `key` 属性。

`jsx` 中不同的元素类型，有不同的 `type` ：

```ini 复制代码
<Component name="xxxx" />  //  type = Component, 是一个函数
<div></div>    // type = "div", 是一个字符串
<React.Fragment></React.Fragment>  // type = React.Fragment, 是一个数字(react 内部定义的);
```

`jsx` 模板转化为 `react element` 以后， `react element` 的 `type` 属性会根据 `jsx` 元素的类型赋不同的值，可能是组件函数，也可能是 `dom` 标签字符串，还可能是数字。 `react element` 转化为 `fiber node` 以后， `react element` 的 `type` 属性也会作为 `fiber node` 的 `type` 属性。

综上，判断拷贝 `current fiber node` 的逻辑，概括来就是：

```
go 复制代码
reactElement.key === currentFiberNode.key && reactElement.type === currentFiberNode.type, current fi

reactElement.key !== currentFiberNode.key, current fiber node //不可克隆；

reactElement.key === currentFiberNode.key && reactElement.type !== currentFiberNode.type, current fi
```

diff 算法：

- 已匹配的父节点的直接子节点进行比较，不跨父节点比较；
- 通过比较 `key` 、 `type` 来判断是否需要克隆 `current fiber node` 。只有 `key` 和 `type` 都相等，才克隆 `current fiber node` 作为新的节点，否则就需要新建一个节点。 `key` 值和节点类型 `type` ， `key` 的优先级更高。如果 `key` 值不相同，那么节点不可克隆。
- 当比较 `single react element` 和 `current fiber node list` 时，只需要遍历 `current fiber node list` ，比较每个 `current fiber node` 和 `react element` 的 `key` 值和 `type` 。只有 `key` 和 `type` 都相等， `react element` 和 `current fiber node` 才能匹配。如果有匹配的，直接克隆 `current fiber node` ，作为 `react element` 对应的 `workInProgress fiber node` 。如果没有匹配的 `current fiber node` ，就需要为 `react element` 重新创建一个新的 `fiber node` 作为 `workInProgress fiber node` 。
- 当比较 `react element list` 和 `current fiber node list` 时，还需要通过列表下标 `index` 判断 `wokrInProgress fiber node` 是否相对于克隆的 `current fiber node` 发生了移动。这也是 `diff` 中最复杂的地方。

为发生变化的 `fiber node` 标记 `effect`

判断节点是否发生变化：

- 节点只要是重新创建的而不是克隆自 `current fiber node` ，那么节点就百分之百发生了变化，需要更新；
- 节点克隆自 `current fiber node` ，需要比较 `props` 是否发生了变化，如果 `props` 发生了变化，节点需要更新；
- 节点克隆自 `current fiber node` ，且是组件类型，还需要比较 `state` 是否发生了变化，如果 `state` 发生了变化，节点需要更新；

常见的 `effect` 类型：

- `Placement`，放置，只针对 `dom` 类型的 `fiber node`，表示节点需要做移动或者添加操作。
- `Update`，更新，针对所有类型的 `fiber node`，表示 `fiber node` 需要做更新操作。
- `PlacementAndUpdate`，放置并更新，只针对 `dom` 类型的 `fiber node`，表示节点发生了移动且 `props` 发生了变化。
- `Ref`，表示节点存在 `ref`，需要初始化 / 更新 `ref.current`。
- `Deletion`，删除，针对所有类型的 `fiber node`，表示 `fiber node` 需要移除。
- `Snapshot`，快照，主要是针对类组件 `fiber node`。当类组件 `fiber node` 发生了 `mount` 或者 `update` 操作，且定义了 `getSnapshotBeforeUpdate` 方法，就会标记 `Snapshot`。
- `Passive`，主要针对函数组件 `fiber node`，表示函数组件使用了 `useEffect`。当函数组件节点发生 `mount` 或者 `update` 操作，且使用了 `useEffect hook`，就会给 `fiber node` 标记 `Passive`。
- `Layout`，主要针对函数组件 `fiber node`，表示函数组件使用了 `useLayoutEffect`。当函数组件节点发生 `mount` 或者 `update` 操作，且使用了 `useLayoutEffect hook`，就会给 `fiber node` 标记 `Layout`。

react 使用二进制数来声明 `effect`，如 `Placement` 为 2 (0000 0010)，`Update` 为 4 (0000 0100)。一个 `fiber node` 可同时标记多个 `effect`，如函数组件 `props` 发生变化且使用了 `useEffect hook`，那么就可以使用 `Placement | Update = 516(位运算符)` 来标记。

收集带 `effect` 的 `fiber node`

如果一个 `fiber node` 被标记了 `effect`，那么 `react` 就会在这个 `fiber node` 完成协调以后，将这个 `fiber node` 收集到 `effectList` 中。当整颗 `fiber tree` 完成协调以后，所有被标记 `effect` 的 `fiber node` 都被收集到一起。

收集 `fiber node` 的 `effectList` 采用单链表结构存储，`firstEffect` 指向第一个标记 `effect` 的 `fiber node`，`lastEffect` 标记最后一个 `fiber node`，节点之间通过 `nextEffect` 指针连接。

由于 `fiber tree` 协调时采用的顺序是深度优先，协调完成的顺序是子节点、子节点兄弟节点、父节点，所以收集带 `effect` 标记的 `fiber node` 时，顺序也是子节点、子节点兄弟节点、父节点。

Render

`render` 也称为 `commit`，是对协调过程中标记的 `effect` 的处理

`effect` 的处理分为三个阶段，这三个阶段按照从前到后的顺序为：

1. `before mutation` 阶段 (`dom` 操作之前)

2. `mutation` 阶段 (`dom` 操作)

3. `layout` 阶段 (`dom` 操作之后)

不同的阶段，处理的 `effect` 种类也不相同。在每个阶段，react 都会从 `effectList` 链表的头部 - `firstEffect` 开始，按序遍历 `fiber node`，直到 `lastEffect`。

before mutation 阶段

`before mutation` 阶段的主要工作是处理带 `Snapshot` 标记的 `fiber node`。 从 `firstEffect` 开始遍历 `effect` 列表，如果 `fiber node` 带 `Snapshot` 标记，触发 `getSnapshotBeforeUpdate` 方法。

mutation 阶段

`mutation` 阶段的主要工作是处理带 `Deletion` 、 `Placement`、 `PlacementAndUpdate`、 `Update` 标记的 `fiber node`。 在这一阶段，涉及到 `dom` 节点的更新、新增、移动、删除，组件节点删除导致的 `componentWillUnmount`、 `destory` 方法的触发，以及删除节点引发的 `ref` 引用的重置。

`dom` 节点的更新：

- 通过原生的 API `setAttribute`、 `removeArrribute` 修改 `dom` 节点的 `attr`；
- 直接修改 `dom` 节点的 `style`；
- 直接修改 `dom` 节点的 `innerHtml`、 `textContent`；

`dom` 节点的新增和移动：

- 如果新增(移动)的节点是父节点的最后一个子节点，那么可以直接使用 `appendChild` 方法。
- 如果不是最后一个节点，需要使用 `insertBefore` 方法。通过遍历找到第一个没有带 `Placement` 标记的节点作为 `insertBefore` 的定位元素。

`dom` 节点的删除：

- 如果节点是 `dom` 节点，通过 `removeChild` 移除；
- 如果节点是组件节点，触发 `componentWillUnmount` 、 `useEffect` 的 `destory` 方法的执行；
- 如果标记 `Deletion` 的节点的子节点中有组件节点，深度优先遍历子节点，依次触发子节点的 `componentWillUnmount` 、 `useEffect` 的 `destory` 方法的执行；
- 如果标记 `Deletion` 的节点及子节点关联了 `ref` 引用，要将 `ref` 引用置空，及 `ref.current = null` (也是深度优先遍历)；

layout 阶段

`layout` 阶段的主要工作是处理带 `update` 标记的组件节点和带 `ref` 标记的所有节点。 工作内容如下：

- 如果类组件节点是 `mount` 操作，触发 `componentDidMount` ；如果是 `update` 操作，触发 `componentDidUpdate` ；
- 如果函数组件节点时 `mount` 操作，触发 `useLayoutEffect` 的 `callback` ；如果是 `update` 操作，先触发上一次更新生成的 `destory` ，再触发这一次的 `callback` ；
- 异步调度函数组件的 `useEffect` ；
- 如果组件节点关联了 `ref` 引用，要初始化 `ref.current` ；