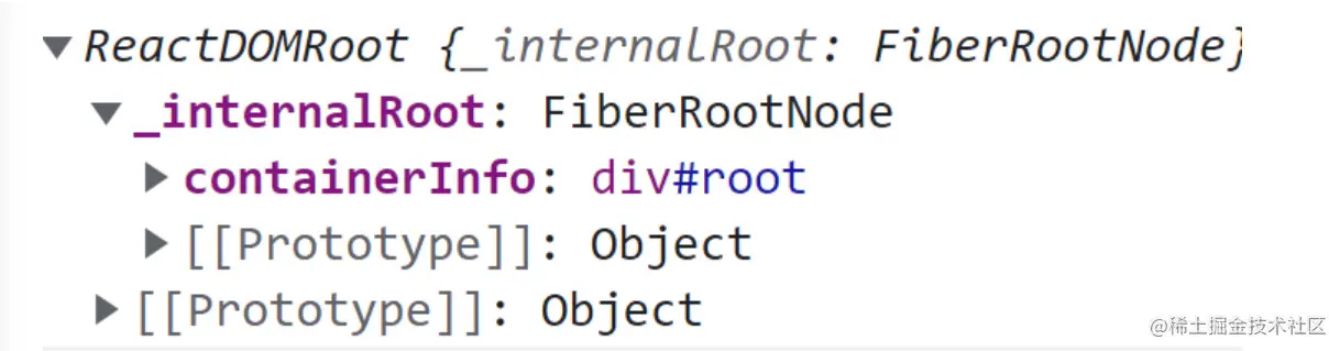
# [React 源码] React 18.2 - 初次并发渲染 [1.8k 字 - 阅读时长 7min]

问题：下面这段 React 代码的初次渲染流程是怎样的?

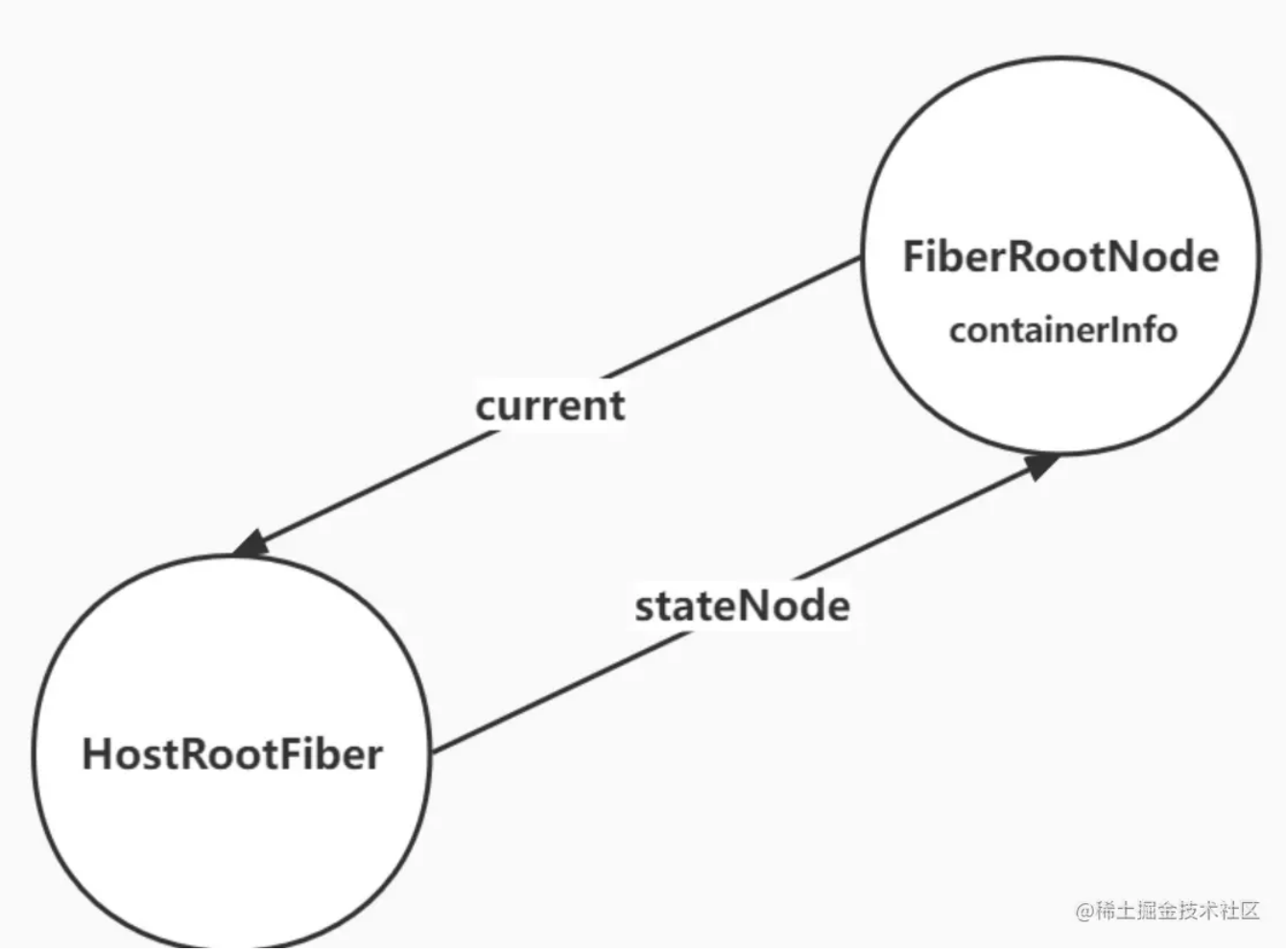
```javascript
let App = <h1>Hello World</h1>;
const root = createRoot(document.getElementById("root"));

root.render(<App />);
```

第一：调用 `createRoot` 函数, 传入 `root` 节点。创建 FiberRoot 节点。




```javascript
export function createRoot(
  container: Element | Document | DocumentFragment,
  options?: CreateRootOptions
): RootType {
  const root = createContainer(
    container,
    ConcurrentRoot,
    null,
    isStrictMode,
    concurrentUpdatesByDefaultOverride,
    identifierPrefix,
    onRecoverableError,
    transitionCallbacks
  );

  return new ReactDOMRoot(root);
}
```

第二：调用 `render` 函数，在 `updateContainer` 函数当中 创建 `RootFiber` 节点。与 `FiberRoot` 建立如下图所示的关系。




```javascript
ReactDOMHydrationRoot.prototype.render = ReactDOMRoot.prototype.render =
  function (children: ReactNodeList): void {
    const root = this._internalRoot;
    updateContainer(children, root, null, null);
  };
```

第三：在 `updateContainer` 函数中，创建 根节点的 update 对象。调用 `enqueueUpdate函数` 入队。

```javascript
export function updateContainer(
  element: ReactNodeList,
  container: OpaqueRoot,
  parentComponent: ?React$Component<any, any>,
  callback: ?Function
): Lane {
  const update = createUpdate(eventTime, lane);
  // 根节点的 update 对象 element 就是 App函数的虚拟 Dom
  update.payload = { element };
```

```
  const root = enqueueUpdate(current, update, lane);

  if (root !== null) {
    scheduleUpdateOnFiber(root, current, lane, eventTime);
    entangleTransitions(root, current, lane);
  }

  return lane;
}
```

第四：然后再 `updateContainer` 函数中调用 `scheduleUpdateOnFiber` -> `ensureRootIsScheduled` 函数

javascript 复制代码

```
export function scheduleUpdateOnFiber(
  root: FiberRoot,
  fiber: Fiber,
  lane: Lane,
  eventTime: number
) {
  ensureRootIsScheduled(root, eventTime);
}
```

第五： `ensureRootIsScheduled` 函数，以一个 `schedulerPriorityLevel` （初次渲染，优先级是 `16` ）优先级开始调度执行 `performConcurrentWorkOnRoot` 。 `Scheduler` 返回一个 `newCallbackNode` 赋值给 根 Fiber 的 `callbackNode` 属性。 这是可以并发的关键。

> `schedulerPriorityLevel` 是根据 `lanesToEventPriority(nextLanes)` 函数，将 `React lane` 模型中的优先级转换成了 `React 事件优先级` ， 最后再转换成 `React Scheduler 优先级` 。为什么要转换？因为 `React Scheduler` 只有 5个优先级，但是 `React lane` 模型有 31 个 优先级，要想用这 5 个 Scheduler 优先级调度 这 31 个 lane 优先级的任务，就应该先将 31 lane 合并成 5 个 `React 事件优先级` ， 最后转换成 5 个 `React Scheduler 优先级` 。

javascript 复制代码

```
function ensureRootIsScheduled(root: FiberRoot, currentTime: number) {
  newCallbackNode = scheduleCallback(
    schedulerPriorityLevel,
    performConcurrentWorkOnRoot.bind(null, root)
  );
  root.callbackPriority = newCallbackPriority;
  root.callbackNode = newCallbackNode;
}
```

第六：在浏览器一帧的空闲时间中（5ms），`performConcurrentWorkOnRoot` 函数被调度执行。这里先会判断是否有条件去进行 `时间切片`。如果应该时间切片则并发渲染，那么调用 `renderRootConcurrent` 函数, 否则,同步渲染调用 `renderRootSync` 函数。

初次渲染，走时间切片并发渲染。 `renderRootConcurrent` 函数。因为初次渲染优先级是 16，不是同步优先级，时间片没有过期，不包括阻塞线程任务。

```javascript 复制代码
function performConcurrentWorkOnRoot(root, didTimeout) {
  const shouldTimeSlice =
    !includesBlockingLane(root, lanes) &&
    !includesExpiredLane(root, lanes) &&
    (disableSchedulerTimeoutInWorkLoop || !didTimeout);
  let exitStatus = shouldTimeSlice
    ? renderRootConcurrent(root, lanes)
    : renderRootSync(root, lanes);

  ensureRootIsScheduled(root, now());

  return null;
}
```

第七： `renderRootConcurrent` 函数中去 调用 `workLoopSync` 函数。while 循环中 调用 `performUnitOfWork` 函数。

```javascript 复制代码
function workLoopConcurrent() {
  // Perform work until Scheduler asks us to yield
  while (workInProgress !== null && !shouldYield()) {
    // $FlowFixMe[incompatible-call] found when upgrading Flow
    performUnitOfWork(workInProgress);
  }
}

function renderRootConcurrent(root: FiberRoot, lanes: Lanes) {
  workLoopConcurrent();
}
```

第八： `performUnitOfWork` 函数 当中就是我们熟悉的 `beginWork` `completeWork`

```javascript 复制代码
function performUnitOfWork(unitOfWork: Fiber): void {
  const current = unitOfWork.alternate;
  setCurrentDebugFiberInDEV(unitOfWork);
```

```
  let next;
  if (enableProfilerTimer && (unitOfWork.mode & ProfileMode) !== NoMode) {
    startProfilerTimer(unitOfWork);
    next = beginWork(current, unitOfWork, renderLanes);
    stopProfilerTimerIfRunningAndRecordDelta(unitOfWork, true);
  } else {
    next = beginWork(current, unitOfWork, renderLanes);
  }

  resetCurrentDebugFiberInDEV();
  unitOfWork.memoizedProps = unitOfWork.pendingProps;
  if (next === null) {
    // If this doesn't spawn new work, complete the current work.
    completeUnitOfWork(unitOfWork);
  } else {
    workInProgress = next;
  }

  ReactCurrentOwner.current = null;
}
```

第九：进入 `beginWork` 初次挂载阶段， 根据不同的标签去对应挂载不同的 mount 逻辑。

javascript 复制代码

```
function beginWork(
  current: Fiber | null,
  workInProgress: Fiber,
  renderLanes: Lanes
): Fiber | null {
  switch (workInProgress.tag) {
    case IndeterminateComponent: {
      return mountIndeterminateComponent(
        current,
        workInProgress,
        workInProgress.type,
        renderLanes
      );
    }
    case FunctionComponent: {
      const Component = workInProgress.type;
      const unresolvedProps = workInProgress.pendingProps;
      const resolvedProps =
        workInProgress.elementType === Component
          ? unresolvedProps
          : resolveDefaultProps(Component, unresolvedProps);
      return updateFunctionComponent(
        current,
        workInProgress,
```

```
        Component,
        resolvedProps,
        renderLanes
      );
    }
  }
}
```

第十：在不同的挂载逻辑当中执行 `reconcilChildren` 构建 子 fiber。初次渲染走的是，`mountChildFibers` 函数，任务就是构建子fiber树。 juejin.cn/post/715210... 这篇文章有些 关于 fiber树的问题。

javascript 复制代码

```javascript
export function reconcileChildren(
  current: Fiber | null,
  workInProgress: Fiber,
  nextChildren: any,
  renderLanes: Lanes
) {
  if (current === null) {
    workInProgress.child = mountChildFibers(
      workInProgress,
      null,
      nextChildren,
      renderLanes
    );
  } else {
    workInProgress.child = reconcileChildFibers(
      workInProgress,
      current.child,
      nextChildren,
      renderLanes
    );
  }
}
```

第十一：当前工作单元的 `fiber` 的 `reconciler` 完毕, 假设 5ms 的时间片到期 退出 while 循环。 退出 `workLoopConcurrent` 函数。

javascript 复制代码

```javascript
    newCallbackNode = scheduleCallback(
      schedulerPriorityLevel,
      performConcurrentWorkOnRoot.bind(null, root),
    );
  }
```

```
  root.callbackPriority = newCallbackPriority;
  root.callbackNode = newCallbackNode;
```

第十二： 返回到 `performConcurrentWorkOnRoot` 函数中，由于 `reconciler` 过程还未完成，`root.callbackNode === originalCallbackNode` 成立, 给 Schduler 返回 `performConcurrentWorkOnRoot` 函数， Schduler 判断是函数之后，不会将该任务弹出优先级队列，而是继续下一帧时间继续执行 `performConcurrentWorkOnRoot`，直至 构建完整棵 fiber 树，返回 null, `Schduler` 判断是 null ，之后将该任务弹出优先级队列。

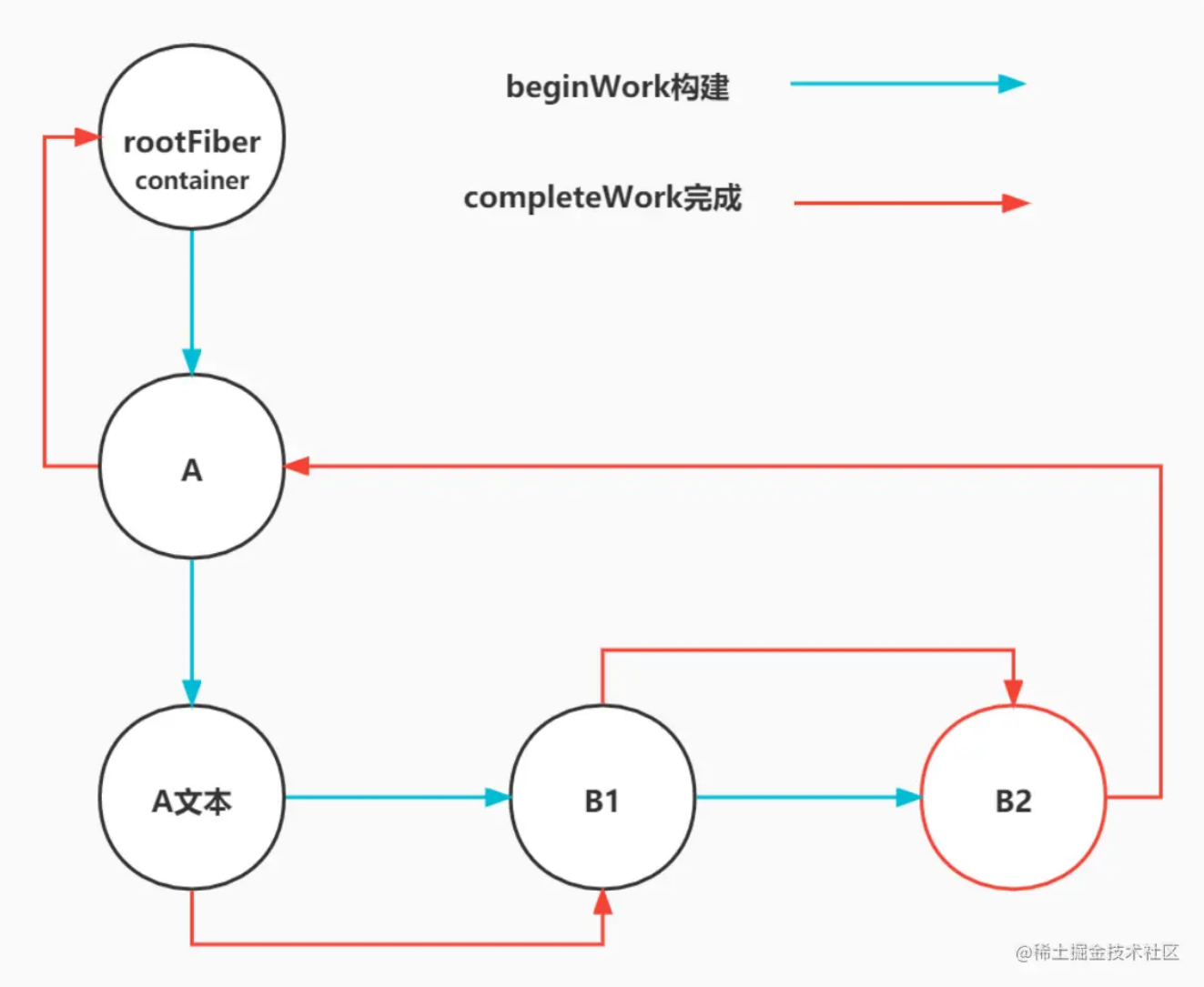
javascript 复制代码

```javascript
  if (root.callbackNode === originalCallbackNode) {
    if (
      workInProgressSuspendedReason === SuspendedOnData &&
      workInProgressRoot === root
    ) {
      root.callbackPriority = NoLane;
      root.callbackNode = null;
      return null;
    }
    return performConcurrentWorkOnRoot.bind(null, root);
  }

  return null;
```

第十三：fiber 节点的 `beginwork` 结束后会走 `completeUnitOfWork` -> `completeWork` 函数

javascript 复制代码

```javascript
function performUnitOfWork(unitOfWork: Fiber): void {
  next = beginWork(current, unitOfWork, renderLanes);
  if (next === null) {
    completeUnitOfWork(unitOfWork);
  }

  ReactCurrentOwner.current = null;
}
```




`completeWork` 函数中 还是判断不同的标签，进行 不同的 `complete` 逻辑, 我们这里用原生标签来举例。 对于 `HostComponent` 来说， `complete` 时 会调用 `updateHostComponent` 函数

> React 18.2 之后， 大家在 completeWork 阶段 熟悉的 effectsLists 都被删掉了，都改成了 bubbleProperties 函数。这个我们放到更新的文章里面去说。

javascript 复制代码

```
function completeWork(
  current: Fiber | null,
  workInProgress: Fiber,
  renderLanes: Lanes
): Fiber | null {

    case HostComponent: {
    popHostContext(workInProgress);
    const type = workInProgress.type;
    if (current !== null && workInProgress.stateNode != null) {
      updateHostComponent(current, workInProgress, type, newProps);

      if (current.ref !== workInProgress.ref) {
```

```
        markRef(workInProgress);
      }
    } else {
      if (!newProps) {
        // This can happen when we abort work.
        bubbleProperties(workInProgress);
        return null;
      }
    }
  }
}
```

第十四： `updateHostComponent` 函数当中挂载 `props` , 并且通过 `appendAllChildren` 函数 将所有子节点 `DOM` 全部插入到自己的 DOM 当中。这样也就意味着，进行最后的提交阶段时，只需要将 App 函数组件的第一个 child 插入到 div#root 容器中就可以完成挂载。因为其余节点在 `completeWork` 函数中就已经插入到了父 DOM 当中去了。

javascript 复制代码

```
updateHostComponent = function (
  current: Fiber,
  workInProgress: Fiber,
  type: Type,
  newProps: Props
) {
  appendAllChildren(newInstance, workInProgress, false, false);
};
```

第十四：在 `commit` 时, 判断子树上有副作用 或者 根节点上有副作用，就可以提交，否则不提交。 初次渲染只有 根节点的第一个 child 有副作用，其余没有副作用 ,可以提交。 在 `mutation` 阶段 调用 `commitMutationEffects` 函数。

javascript 复制代码

```
  if (subtreeHasEffects || rootHasEffect) {
    commitMutationEffects(root, finishedWork, lanes);
 }
```

> 为什么其余子节点没有副作用 而根节点的第一个 `child` 有副作用 ？ 因为 `reconcileChildren` 时，其余节点走到是 `mountChildFibers` 函数，而根节点的第一个 `child` 走的是 `reconcileChildFibers` 函数。
>
> 为什么根节点的第一个 `child` 走的是 `reconcileChildFibers` 函数? 这是因为挂载的时候， `rootFiber` 已经有 `current` 属性了。所以 `current !== null` , 走

> `reconcileChildFibers` 函数，并在根节点的第一个 `child` 节点上 追加副作用： `|= PLACEMENT` 。

javascript 复制代码

```
export function reconcileChildren(
  current: Fiber | null,
  workInProgress: Fiber,
  nextChildren: any,
  renderLanes: Lanes
) {
  if (current === null) {
    workInProgress.child = mountChildFibers(
      workInProgress,
      null,
      nextChildren,
      renderLanes
    );
  } else {
    workInProgress.child = reconcileChildFibers(
      workInProgress,
      current.child,
      nextChildren,
      renderLanes
    );
  }
}
```

`commitMutationEffects` 函数中，通过 `recursivelyTraverseMutationEffects` 函数，递归提交子节点的作用， `commitReconciliationEffects` 函数递归提交自己的副作用。 将有副作用的的节点，根据副作用标签进行增删改查。比如本例当中：从根节点开始递归提交，递归到 根节点的第一个 child `<h1></h1>` 时， 有副作用 `PLACEMENT` 则调用 `commitPlacement(finishedWork)` 函数 通过 `appendChild()` 插入到根节点当中，完成挂载。

javascript 复制代码

```
function commitMutationEffectsOnFiber(
  finishedWork: Fiber,
  root: FiberRoot,
  lanes: Lanes
) {
  const current = finishedWork.alternate;
  const flags = finishedWork.flags;
  switch (finishedWork.tag) {
    // eslint-disable-next-line-no-fallthrough
    case HostComponent: {
      recursivelyTraverseMutationEffects(root, finishedWork, lanes);
```

```
        commitReconciliationEffects(finishedWork)
}
```

> 如果是根节点的第一个 child 是一个 App 函数/类/Memo组件，则又是不同的提交逻辑……

## 自此完成挂载