React 18源码解析（二）—— 初始化渲染

该部分解析基于我们实现的简单版react18中的代码，是react18源码的阉割版，希望用最简洁的代码来了解react的核心原理。其中大部分逻辑和结构都和源码保持一致，方便阅读源代码。

上一章节末尾提到了performSyncWorkOnRoot方法，这个方法是同步调度的入口函数，其内部有两大核心流程：

- render 阶段（renderRootSync）：这个阶段主要做的事情就是构建/更新 fiber 树
- commit 阶段（commitRoot）：这个阶段会进行一系列副作用、DOM 相关的操作

```
// react-reconciler/src/ReactFiberWorkLoop.ts
function performSyncWorkOnRoot(root: FiberRoot) {
  // 省略其它代码
  renderRootSync(root, lanes);
  // 省略其它代码
  commitRoot(root);
  return null;
}
```

一、render阶段

不管是同步的 render 还是并发的 render，最终都会执行 performUnitOfWork，两者的区别只在于并发渲染可能会被中断，而同步渲染不会被中断。

```
// react-reconciler/src/ReactFiberWorkLoop.ts
// 并发渲染
function workLoopConcurrent() {
  // 留给react render的时间片不够就会中断render
  while (workInProgress !== null && !shouldYield()) {
    performUnitOfWork(workInProgress);
  }
}

// 同步渲染
function workLoopSync() {
  // 对于已经超时（同步）的任务，不需要检查是否需要yield，直接执行
  while (workInProgress !== null) {
    performUnitOfWork(workInProgress);
  }
}
```

在执行真正的渲染工作 performUnitOfWork 之前，还需要有一个准备工作 prepareFreshStack

```
// react-reconciler/src/ReactFiberWorkLoop.ts
// 同步模式
function renderRootSync(root: FiberRoot, lanes: Lanes) {
  // 根应用节点或者优先级改变，优先级改变是因为高优先级任务的插入打断了上一次低优先级任务的执行
  if (workInProgressRoot !== root || workInProgressRootRenderLanes !== lanes) {
    // 为接下去新一次渲染工作初始化参数，清除上一次渲染已经产生的工作
    prepareFreshStack(root, lanes);
  }
  do {
```

```
      try {
        workLoopSync();
        break;
      } catch (error: any) {
        throw Error(error);
      }
    } while (true);
    // 省略其它代码
  }

  // 并发模式
  function renderRootConcurrent(root, lanes: Lanes) {
    // 如果根应用节点或者优先级改变，则创建一个新的workInProgress
    if (workInProgressRoot !== root || workInProgressRootRenderLanes !==
  lanes) {
      // 为接下去新一次渲染工作初始化参数
      prepareFreshStack(root, lanes);
    }
    do {
      try {
        workLoopConcurrent();
        break;
      } catch (e: any) {
        throw Error(e);
      }
    } while (true);
    // 省略其它代码
  }
```

prepareFreshStack 的作用是为接下去新一次渲染工作初始化参数并清除上一次渲染已经产生的工作 —— workInProgress

React Fiber 采用了一种叫做双缓存的架构。双缓存是一种常见的图形显示技术，它有两个区域，一个区域用于显示已经渲染好的图像，另一个区域用于缓存在内存中更新的图像。当图像更新完成后，将更新结果复制到显示区域中，这种方式可以有效避免复杂更新带来的页面闪烁问题。简单理解就是不会看到绘制过程，直接就能看到结果。

在 React 中，页面上显示的 fiber 叫做 current fiber，在内存中更新的 fiber 叫做 workInProgress fiber，两者通过 alternate 指针进行对应。在上一章节有讲到，整个应用的根节点有一个 current 指针指向当前应用的根节点，这个 current 指针指向的 fiber 就是已经显示到页面上的 fiber。当页面发起更新时，会基于 current fiber 构建 workInProgress fiber，当 workInProgress fiber 更新完成时，current 指针会指向构建好的 workInProgress fiber，从而实现双缓存更新。

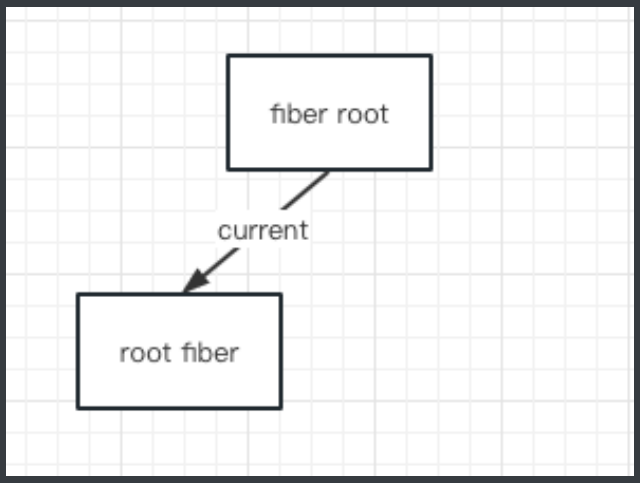
```
// react-reconciler/src/ReactFiberWorkLoop.ts
function prepareFreshStack(root: FiberRoot, lanes: Lanes) {
  // 省略其它代码
  const rootWorkInProgress = createWorkInProgress(root.current, null);
  workInProgress = rootWorkInProgress;
  // 省略其它代码
  return workInProgressRoot;
}

// react-reconciler/src/ReactFiber.ts
export function createWorkInProgress(current, pendingProps) {
  let workInProgress = current.alternate;
  if (workInProgress === null) {
    workInProgress = createFiber(current.tag, pendingProps,
current.key);
    workInProgress.elementType = current.elementType;
    workInProgress.type = current.type;
    workInProgress.stateNode = current.stateNode;
    workInProgress.alternate = current;
    current.alternate = workInProgress;
  } else {
    workInProgress.pendingProps = pendingProps;
    // 复用current的一些属性值
    workInProgress.type = current.type;
    // 重置flags、subtreeFlags、deletions
    workInProgress.flags = NoFlags;
    workInProgress.subtreeFlags = NoFlags;
    workInProgress.deletions = null;
  }
```

```
    // 省略其它代码
    workInProgress.child = current.child;
    workInProgress.sibling = current.sibling;
    workInProgress.index = current.index;
    workInProgress.memoizedProps = current.memoizedProps;
    workInProgress.memoizedState = current.memoizedState;
    workInProgress.updateQueue = current.updateQueue;

    workInProgress.sibling = current.sibling;
    workInProgress.index = current.index;

    return workInProgress;
  }
```

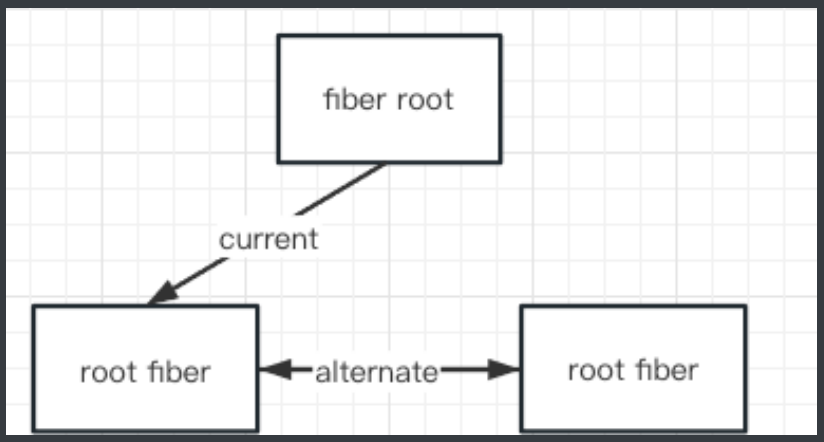
这里以 App组件为例大致介绍一下双缓存的构建过程：

```
const App = () =>{
  return <div><span></span></div>;
}
const root = ReactDOM.createRoot(document.getElementById('app'));
root.render(<App />);
```

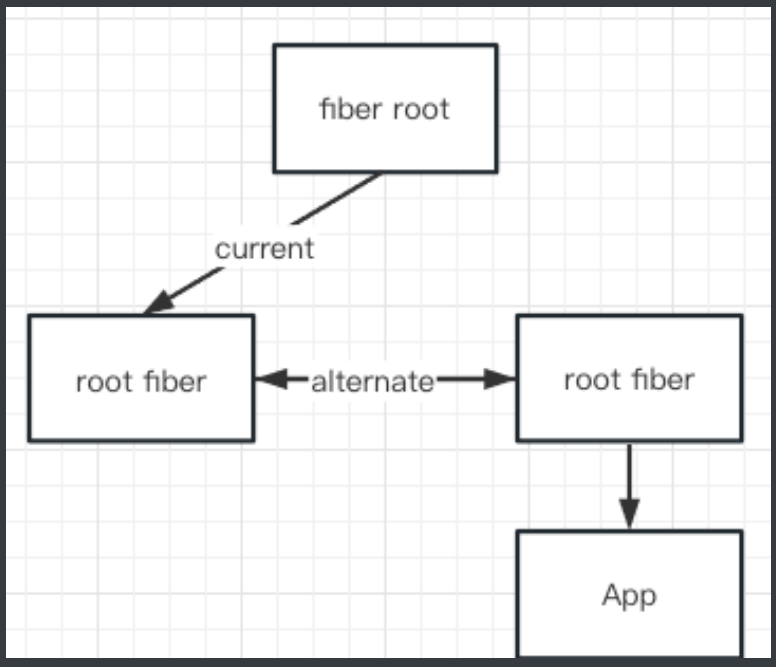
1. 在 prepareFreshStack 之前已经存在了FiberRoot 和 RootFiber，FiberRoot 的 current 指针指向当前应用的根节点 RootFiber



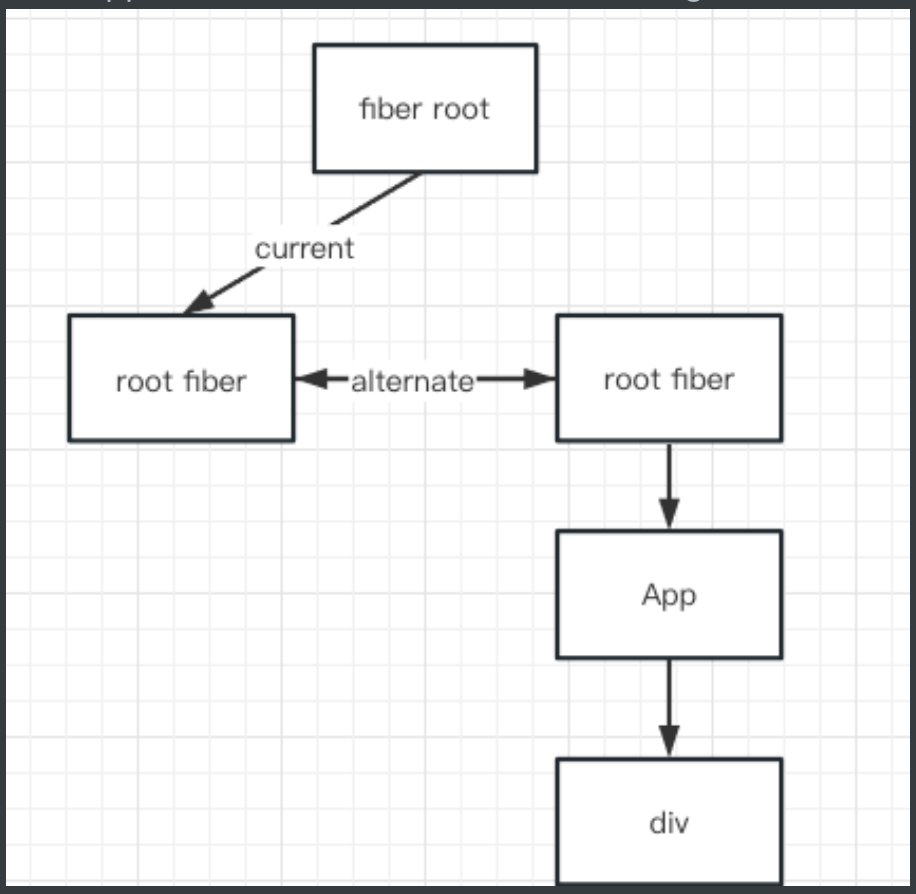
2. 在第一次执行 prepareFreshStack 时会先创建 root fiber 对应的 workInProgress fiber，两者通过 alternate 进行关联



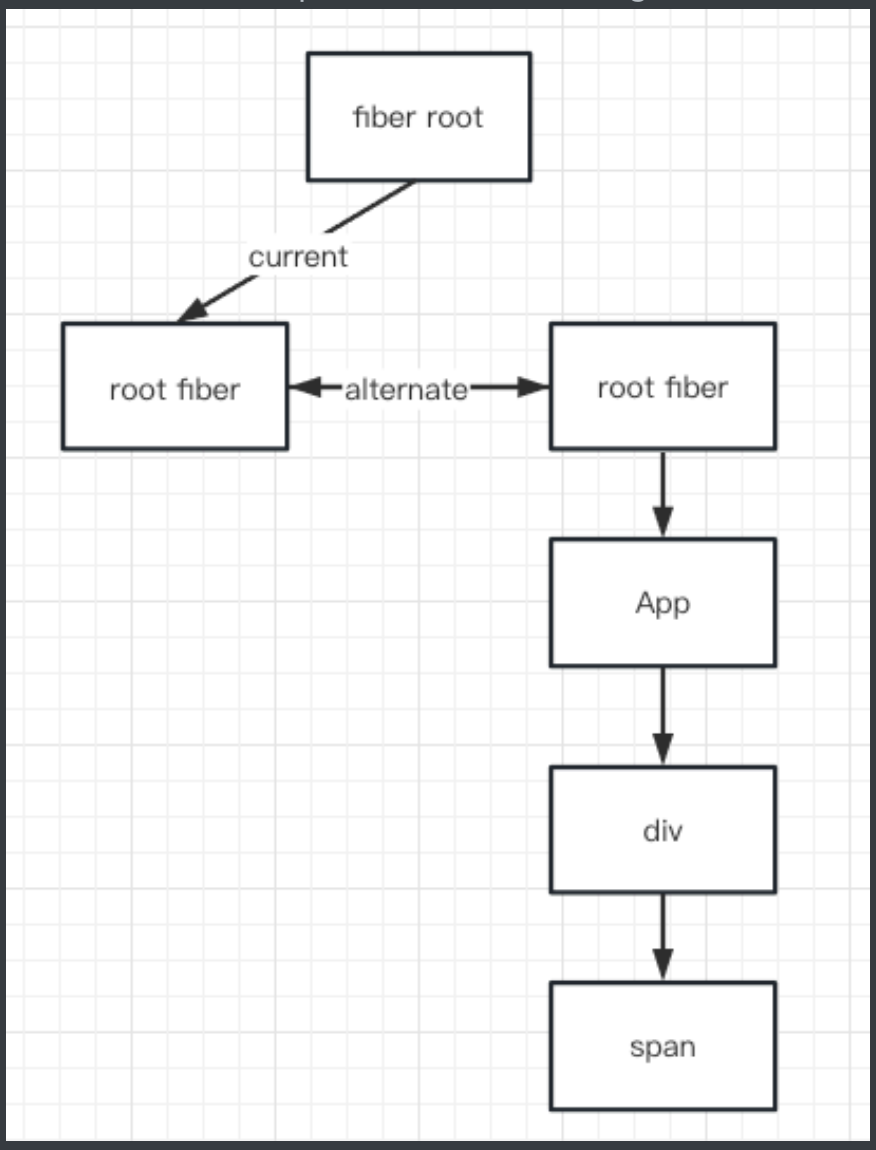
3. 接下去会渲染 App 组件，创建 App 组件对应的 workInProgress fiber



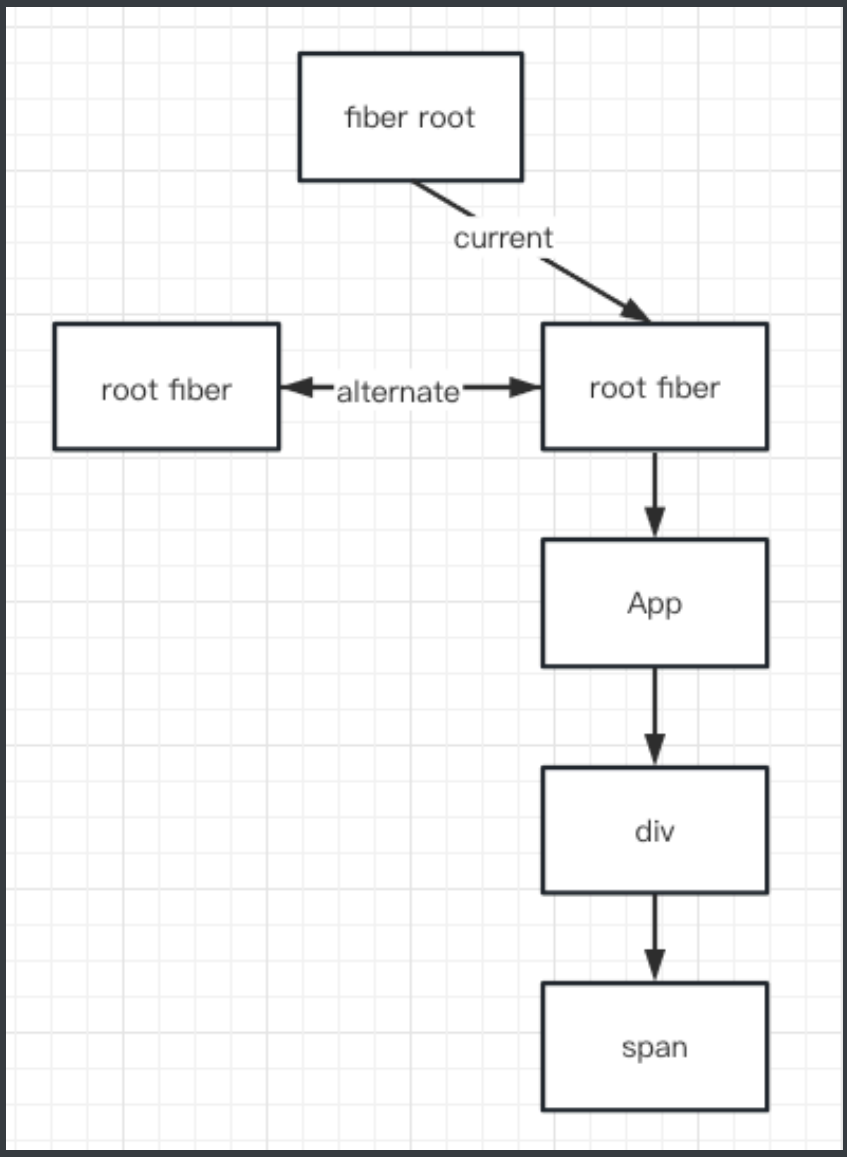
4. 创建 App 组件的子节点 div 对应的 workInProgress fiber



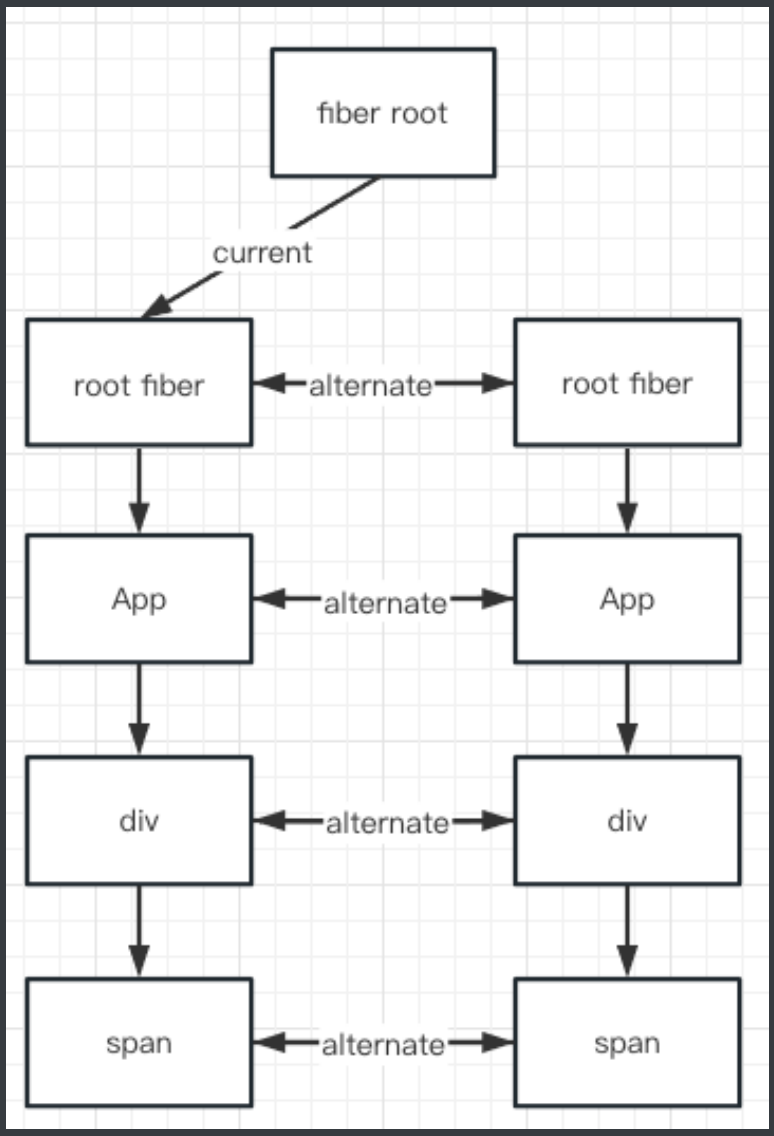
5. 创建 div 的子节点 span 对应的 workInProgress fiber



6. 页面更新完成后 current 指针指向右边的 workInProgress fiber 树



7. 当组件更新时，会基于 current fiber 创建一颗新的 workInProgress fiber 树



了解 react 双缓存的基本概念后，我们开始进入渲染构建流程 performUnitOfWork。在 performUnitOfWork 中有两个方法 beginWork 和 completeUnitOfWork，这是渲染阶段中的两个重要阶段，一个称为递阶段，另一个称为归阶段。beginWork 执行完会返回一个 fiber 节点，如果这个节点存在，则该节点会进入下一个 beginWork 的处理流程；如果不存在，则说明该条链路上的节点已经处理完成，开始 completeUnitOfWork 阶段。

```
// react-reconciler/src/ReactFiberWorkLoop.ts
/**
 * @description: 以fiber节点为单位开始beginWork和completeWork
 * @param unitOfWork
 */
```

```
function performUnitOfWork(unitOfWork: Fiber) {
  // 首屏渲染只有当前应用的根结点存在current，其它节点current为null
  const current = unitOfWork.alternate;
  const next = beginWork(current, unitOfWork, subtreeRenderLanes);
  // 属性已经更新到dom上了，memoizedProps更新为pendingProps
  unitOfWork.memoizedProps = unitOfWork.pendingProps;

  // 不存在子fiber节点了，说明节点已经处理完，此时进入completeWork
  if (next == null) {
    completeUnitOfWork(unitOfWork);
  } else {
    workInProgress = next;
  }
}
```

递阶段始于 beginWork，这个方法接受三个参数：

- current：页面中的fiber
- workInProgress：内存中的fiber
- renderLanes：渲染优先级

这个阶段主要做的事情就是采用深度优先遍历的方式创建/更新 fiber 节点。如果 current 节点存在，说明可能是更新阶段或者是首次渲染时的 hostRoot节点（当前应用的根节点），反之就是首次渲染阶段。

```
// react-reconciler/src/ReactFiberBeginWork.ts
export function beginWork(
  current: Fiber | null,
  workInProgress: Fiber,
  renderLanes: Lanes
) {
  // 首屏渲染只有hostRoot存在current节点，其他节点还未被创建
  // hostRoot的workInPgress树中的HostRoot是在prepareFreshStack函数中创建
  if (current !== null) {
    // update阶段，可以复用current（即旧的fiber节点）
```

```
    const oldProps = current.memoizedProps;
    const newProps = workInProgress.pendingProps;
    if (oldProps !== newProps) {
      // 属性更新，标记didReceiveUpdate为true，说明这个fiber需要继续beginWork
的其它工作
      didReceiveUpdate = true;
    } else {
      // 省略其它代码
      didReceiveUpdate = false;
    }
  } else {
    // mount阶段
    didReceiveUpdate = false;
  }
  //
  switch (workInProgress.tag) {
    case HostRoot: {
      // 省略其它代码
    }
    case IndeterminateComponent:
      // 省略其它代码
    case ClassComponent: {
      // 省略其它代码
    }
    case HostComponent:
      // 省略其它代码
    case HostText:
      // 省略其它代码
    case FunctionComponent:
      // 省略其它代码
    case Fragment:
      // 省略其它代码
    case MemoComponent: {
      // 省略其它代码
    }
    case SimpleMemoComponent: {
```

```
        // 省略其它代码
      }
    }
    return null;
  }
```

对于首次渲染/更新阶段，会根据当前传入的 workInProgress fiber类型走不同的处理逻辑。针对首次渲染，会先进入 hostRoot 类型的处理 updateHostRoot：

1. 克隆一份更新队列到 workInProgress fiber上，保证两者结构一致
2. 处理更新队列中的所有的 update 更新对象，计算新的 state 状态
3. 判断新旧节点是否相同，如果相同则表示不需要再继续处理子节点，否则就会继续协调子节点

```
// react-reconciler/src/ReactFiberBeginWork.ts
case HostRoot: {
    return updateHostRoot(current, workInProgress, renderLanes);
}

function updateHostRoot(
  current: Fiber | null,
  workInProgress: Fiber,
  renderLanes: Lanes
) {
  const nextProps = workInProgress.pendingProps;
  const prevState = workInProgress.memoizedState;
  const prevChildren = prevState.element;

  // 克隆一份UpdateQueue
  cloneUpdateQueue(current!, workInProgress);
  processUpdateQueue(workInProgress, nextProps, null, renderLanes);

  const nextState = workInProgress.memoizedState;
  // 获取要更新的jsx元素
  const nextChildren = nextState.element;
```

```
    // 新旧jsx对象没变，直接返回
    if (nextChildren === prevChildren) {
      return null;
    }

    // 创建子fiber节点
    reconcileChildren(current, workInProgress, nextChildren, renderLanes);

    // 返回子fiber节点
    return workInProgress.child;
  }
```

processUpdateQueue 的逻辑比较复杂，我们先简单理解一下这个方法的内部逻辑：

1. 获取 firstBaseUpdate 和 lastBaseUpdate，这里我们不考虑其它情况，只需要把它当作是初始化的 null 就行
2. 从 queue.shared.pending 上获取 pendingQueue，即存放所有 update 对象的环状链表，这里我们需要将环状链表断开变成非环状的状态。这里提一点，react 中需要用到环状链表是为了能更方便地访问到头节点。queue.shared.pending 指向的是最后一个 update 对象，那么第一个 update 就是queue.shared.pending.next。当获取到 firstPendingUpdate 和 lastPendingUpdate 之后，就可以通过将 lastPendingUpdate.next 置为 null 来断开首尾的连接，最终形成非环状的单向链表
3. 如果 current fiber 存在，则同第二步操作，保持两边结构一致
4. 遍历第二步生成的单向链表 firstBaseUpdate，计算新的状态 newState 并将结果存到 memoizedState 上

```
  // react-reconciler/src/ReactUpdateQueue.ts
  export function processUpdateQueue<State>(
    workInProgress: Fiber,
    props: any,
    instance: any,
    renderLanes: Lanes
  ) {
    const queue: UpdateQueue<State> = workInProgress.updateQueue;
```

```
  let firstBaseUpdate = queue.firstBaseUpdate;
  let lastBaseUpdate = queue.lastBaseUpdate;

  // pending始终指向的是最后一个添加进来的Update
  let pendingQueue = queue.shared.pending;

  // 检测shared.pending是否存在进行中的update将他们转移到baseQueue
  if (pendingQueue !== null) {
    queue.shared.pending = null;
    const lastPendingUpdate = pendingQueue;
    // 获取第一个Update
    const firstPendingUpdate = lastPendingUpdate.next;
    // pendingQueye队列是循环的。断开第一个和最后一个之间的指针，使其是非循环的
    lastPendingUpdate.next = null;
    // 将shared.pending上的update接到baseUpdate链表上
    if (lastBaseUpdate === null) {
      firstBaseUpdate = firstPendingUpdate;
    } else {
      firstBaseUpdate = lastBaseUpdate.next;
    }
    lastBaseUpdate = lastPendingUpdate;
    const current = workInProgress.alternate;

    // 如果current也存在，需要将current也进行同样的处理，同fiber双缓存相似
    // Fiber节点最多同时存在两个updateQueue：
    // current fiber保存的updateQueue即current updateQueue
    // workInProgress fiber保存的updateQueue即workInProgress updateQueue
    // 在commit阶段完成页面渲染后，workInProgress Fiber树变为current Fiber
树，workInProgress Fiber树内Fiber节点的updateQueue就变成current
updateQueue。
    if (current !== null) {
      const currentQueue = current.updateQueue;
      const currentLastBaseUpdate = currentQueue.lastBaseUpdate;
```

```
      // 如果current的updateQueue和workInProgress的updateQueue不同，则对
current也进行同样的处理，用于结构共享
      if (currentLastBaseUpdate !== lastBaseUpdate) {
        if (currentLastBaseUpdate === null) {
          currentQueue.firstBaseUpdate = firstPendingUpdate;
        } else {
          currentLastBaseUpdate.next = firstPendingUpdate;
        }
        currentQueue.lastBaseUpdate = lastPendingUpdate;
      }
    }
  }

  if (firstBaseUpdate !== null) {
    let newLanes = NoLanes;

    let newState = queue.baseState;
    let newBaseState: State | null = null;

    let newLastBaseUpdate = null;
    let newFirstBaseUpdate = null;

    let update: Update<State> | null = firstBaseUpdate;
    // TODO 优先级调度
    do {
      newState = getStateFromUpdate(
        workInProgress,
        queue,
        update,
        newState,
        props,
        instance
      );
      // 省略其它代码
      update = update.next;
      if (update === null) break;
```

```
    } while (true);

    // 省略其它代码
    workInProgress.memoizedState = newState;
  }
}
```

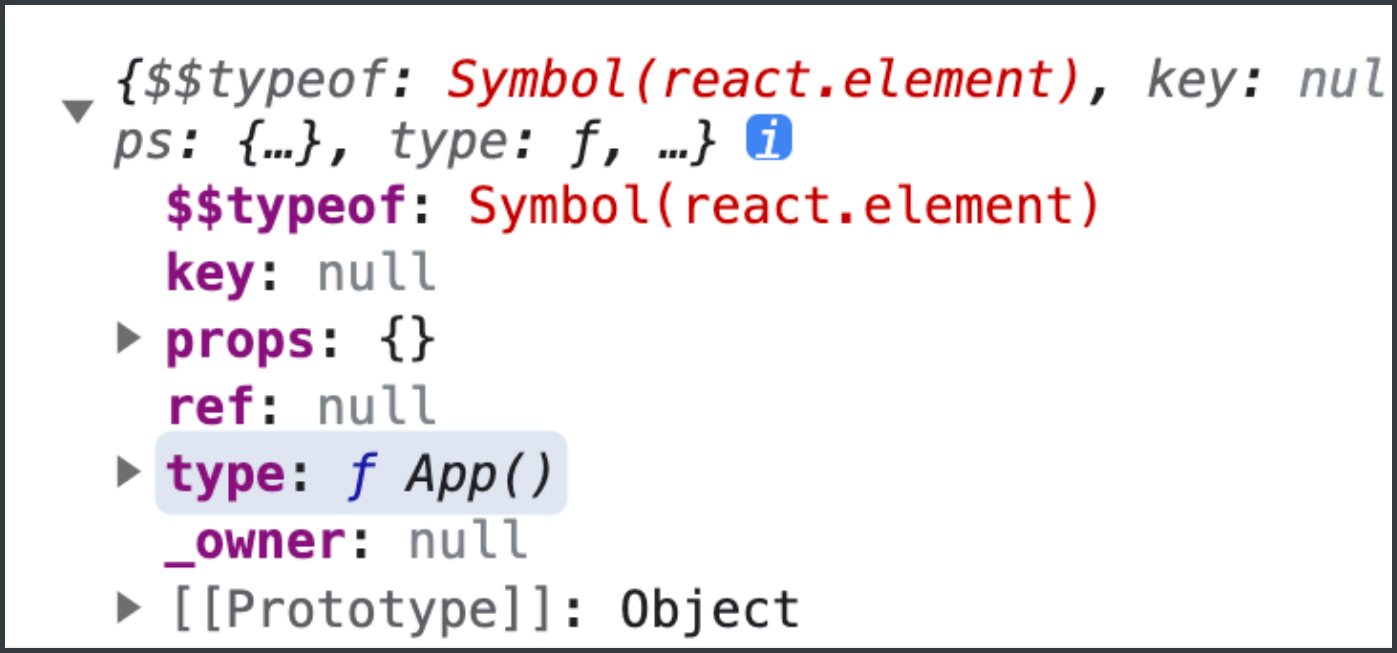
当执行完 processUpdateQueue 之后会比较新旧子节点是否相同，如果相同则不进行子节点的处理，否则会调用 reconcileChildren 处理子节点。reconcileChildren 的逻辑很简单，根据 current fiber 是否存来判断是更新操作还是创建操作。

```
// react-reconciler/src/ReactFiberBeginWork.ts
/**
 * @description: 处理子fiber节点
 */
function reconcileChildren(
  current: Fiber | null,
  workInProgress: Fiber,
  nextChildren: any,
  renderLanes: Lanes
) {
  // current为null说明是首次创建阶段，除了hostRoot节点
  if (current === null) {
    workInProgress.child = mountChildFibers(
      workInProgress,
      null,
      nextChildren,
      renderLanes
    );
  } else {
    // 说明是更新节点，hostRoot节点首次渲染也会进入这里
    workInProgress.child = reconcileChildFibers(
      workInProgress,
      current.child,
      nextChildren,
      renderLanes
```

```
    );
  }
}
```

前面提到，对于首次渲染的 hostRoot 来说 current 是存在的，所以会进入 reconcileChildFibers 方法。我们以 const App = () =>

react
为例先看一下它对应生成的 jsx 对象：



对于 hostRoot 来说，它的第一个子节点是 App 对应的 jsx 对象并且它的 $$typeof 对象是 Symbol(react.element)，因此会进入 reconcileSingleElement 方法，即单子节点的处理。当节点处理完后会通过 placeSingleChild 方法，给节点打上 flag 标记，这个标记会在后面的DOM操作中用到。

```
// demo.jsx
const App = () => <div><span>react</span></div>;

// shared/src/ReactSymbols.ts
export const REACT_ELEMENT_TYPE = Symbol.for("react.element");

// react-reconciler/src/ReactChildFiber.ts
function reconcileChildFibers(
    returnFiber: Fiber,
```

```
    currentFirstChild: Fiber | null,
    newChild: any,
    lanes: Lanes
  ) {
    // 省略其它代码
    if (isObject(newChild)) {
      // 处理单个子节点的情况
      switch (newChild.$$typeof) {
        case REACT_ELEMENT_TYPE:
          return placeSingleChild(
            reconcileSingleElement(
              returnFiber,
              currentFirstChild,
              newChild,
              lanes
            )
          );
      }
    }
    // 省略其它代码
  }
```

在首次渲染的时候，reconcileSingleElement的作用主要是根据 jsx 对象创建对应的 fiber 节点并且通过 return 指针将其和父 fiber 进行关联。在 demo 中，会创建App 组件对应的 fiber 节点。在 fiber 节点创建的过程中会调用 createFiberFromTypeAndProps 方法，顾名思义，这个方法的作用就是根据传入的节点类型还有属性值去创建 fiber 节点。因此，其内部会有很多关于节点类型的判断处理，根据不同的节点类型创建对应的 fiber 节点，不过不管类型如何，最终都会调用 createFiber 来创建 fiber 节点。

在这里 App 组件的 tag 会被默认初始化为 IndeterminateComponent 的类型，表示它还是一个不确定的组件。

```
// react-reconciler/src/ReactChildFiber.ts
 function reconcileSingleElement(
    returnFiber: Fiber,
    currentFirstChild: Fiber | null,
```

```
    element: any,
    lanes: Lanes
  ): Fiber {
    // 省略其它代码
    // 没有节点复用（比如首屏渲染的hostRoot的current是没有child节点的）直接创建
fiber节点
    const created: Fiber = createFiberFromElement(element, lanes);
    created.return = returnFiber;
    return created;
  }

// react-reconciler/src/ReactFiber.ts
export function createFiberFromElement(element: any, lanes: Lanes):
Fiber {
  const { type, key, props } = element;
  const fiber = createFiberFromTypeAndProps(type, key, props, lanes);
  return fiber;
}

export function createFiberFromTypeAndProps(
  type: any,
  key: any,
  pendingProps: any,
  lanes: Lanes
): Fiber {
  let fiberTag: WorkTag = IndeterminateComponent;
  if (isFunction(type)) {
    // 判断是不是class组件
    if (shouldConstruct(type)) {
      fiberTag = ClassComponent;
    }
  } else if (isString(type)) {
    // 说明是普通元素节点
    fiberTag = HostComponent;
  } else {
    // 省略其它代码
```

```
    }

    const fiber = createFiber(fiberTag, pendingProps, key);
    fiber.elementType = type;
    fiber.type = type;
    return fiber;
  }
```

当 fiber 创建完成后会继续调用 placeSingleChild 来给创建好的 fiber 打上 flag 标记，之前创建好的 App 组件的 fiber 会被打上 Placement 的标记，表示这个节点会被放置（DOM添加或者移动）。

```
    // react-reconciler/src/ReactChildFiber.ts
    function placeSingleChild(newFiber: Fiber): Fiber {
      // 首次渲染时的hostRoot节点会进入到这个条件，在更新中新创建的节点也会被打上这个
  标记
      if (shouldTrackSideEffects && newFiber.alternate === null) {
        newFiber.flags |= Placement;
      }
      return newFiber;
    }
```

App 对应的 fiber 处理完成后会作为下一次 beginWork 的入参继续创建其子 fiber。在 createFiberFromTypeAndProps 中 App 组件对应的 tag 类型是 IndeterminateComponent 类型，所以它会进入 mountIndeterminateComponent 这个方法

```
  // react-reconciler/src/ReactFiberBeginWork.ts
  export function beginWork(
    current: Fiber | null,
    workInProgress: Fiber,
    renderLanes: Lanes
  ) {
    // 省略其它代码
    switch (workInProgress.tag) {
      case HostRoot: {
```

```
        return updateHostRoot(current, workInProgress, renderLanes);
      }
      case IndeterminateComponent:
        return mountIndeterminateComponent(
          current,
          workInProgress,
          workInProgress.type,
          renderLanes
        );
       // 省略其它代码
    }
  }
```

mountIndeterminateComponent 内部会调用 renderWithHooks 方法处理函数式组件（省略了 class 组件的处理，这里当 renderWithHook 执行完成后我们默认将其类型改为 FunctionComponent 类型），它返回的 value 值就是组件内部 return 的 jsx 对象。接下去又是创建子 fiber 的过程，这里的 reconcileChildren 第一个参数为 null，即 current 为 null，所以会进入 reconcileChildren 中的 mountChildFibers 方法。

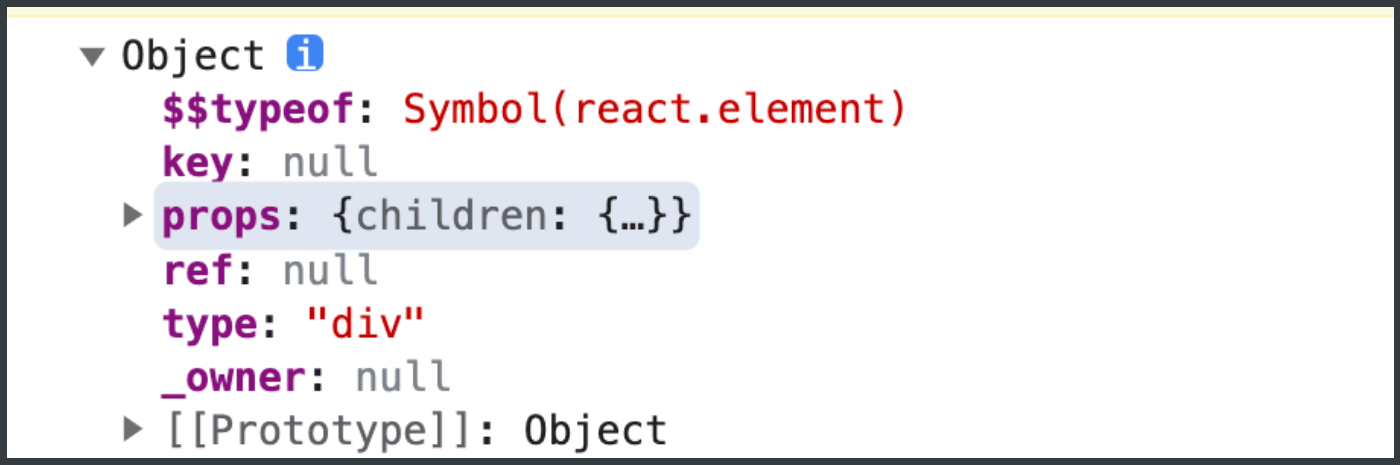
```
  // react-reconciler/src/ReactFiberBeginWork.ts
  // Function组件渲染会进入这里
  function mountIndeterminateComponent(
    _current: Fiber | null,
    workInProgress: Fiber,
    Component: Function,
    renderLanes: Lanes
  ) {
    const props = workInProgress.pendingProps;
    // value值是jsx经过babel处理后得到的vnode对象
    const value = renderWithHooks(
      _current,
      workInProgress,
      Component,
      props,
      null,
      renderLanes
```

```
    );
    // TODO classComponent
    workInProgress.tag = FunctionComponent;
    reconcileChildren(null, workInProgress, value, renderLanes);
    return workInProgress.child;
  }

  // react-reconciler/src/ReactFiberHooks.ts
  export function renderWithHooks(
    current: Fiber | null,
    workInProgress,
    Component: Function,
    props: any,
    secondArg: any,
    nextRenderLanes: Lanes
  ) {
    // 省略其它代码
    // 执行函数式组件方法，返回 jsx 对象
    const children = Component(props, secondArg);
    // 省略其它代码
    return children;
  }
```

mountChildFibers 和之前讲的 reconcileChildFibers 其实都是由 ChildReconciler 函数创建而来，只是根据传入的 shouldTrackSideEffects 标识来区分功能，这个标识暂且可以认为是区分首次渲染和更新。

```
// demo.jsx
const App = () => <div><span>react</span></div>;

// react-reconciler/src/ReactChildFiber.ts
function ChildReconciler(shouldTrackSideEffects) {
  // 省略其它代码
}
export const reconcileChildFibers = ChildReconciler(true);
export const mountChildFibers = ChildReconciler(false);
```

接下去的流程其实和上面 App fiber 的创建流程大致相同了，对于 App 组件返回的 jsx 对象来说，外层对象是 div 节点对应的 jsx 对象，它也是一个 Object 类型，并且 $$typeof 也是 Symbol(react.element)，因此它也会走 reconcileSingleElement 和 placeSingleChild 方法。div 这种普通元素节点和之前讲的 App 组件节点的区别在于创建的 fiber 类型不同，在 createFiberFromTypeAndProps 中，普通元素的 tag 是一个字符串类型，所以它会被标记为 HostComponent 类型。



div fiber 创建完毕后会进入下一个 beginWork 阶段，由于 div 对应的 fiber 是 HostComponent类型，因此它会进入 updateHostComponent 方法。在 updateHostComponent 方法中会先获取子节点，判断子节点是不是唯一文本子节点，唯一文本子节点指的是子节点只有一个文本节点。如果是唯一文本子节点，则直接把 children 置为 null，表示不需要再处理子节点，直接把文本内容当做 props；反之则需要创建子 fiber 节点。显然，div 的子节点是 span 子节点，不是唯一文本子节点。所以接下去会创建 span 节点对应的 fiber，这个过程同父节点 div 相同。

```
// demo.jsx
```

```jsx
const App = () => <div><span>react</span></div>;

// react-reconciler/src/ReactFiberBeginWork.ts
export function beginWork(
  current: Fiber | null,
  workInProgress: Fiber,
  renderLanes: Lanes
) {
  // 省略其它代码
  switch (workInProgress.tag) {
    case HostRoot: {
      return updateHostRoot(current, workInProgress, renderLanes);
    }
    case IndeterminateComponent:
      return mountIndeterminateComponent(
        current,
        workInProgress,
        workInProgress.type,
        renderLanes
      );
    case HostComponent:
      return updateHostComponent(current, workInProgress, renderLanes);
    // 省略其它代码
   }
}

function updateHostComponent(
  current: Fiber | null,
  workInProgress: Fiber,
  renderLanes: Lanes
) {
  const { type, pendingProps: nextProps } = workInProgress;
  let nextChildren = nextProps.children;
  // 判断是否只有唯一文本子节点，这种情况不需要为子节点创建fiber节点
  const isDirectTextChild = shouldSetTextContent(type, nextProps);
  if (isDirectTextChild) {
```

```
      nextChildren = null;
    }
    reconcileChildren(current, workInProgress, nextChildren, renderLanes);
    return workInProgress.child;
  }
```

当 span 节点对应的 fiber 创建完成后会进入下一个 beginWork 阶段，此时需要处理的是 span fiber，span fiber 和 div fiber 的处理相同，只是 span 的子节点只有一个文本子节点 react 。因此 span fiber 只有一个唯一文本子节点，不需要再进行子 fiber 的创建过程，直接返回其 child 属性值，这个值显然是个 null。在 performUnitOfWork 那个方法中提到过，当 beginWork 返回的值是 null 时说明该条链路的节点已经全部处理完毕，接下去会进入归阶段。

```
// react-reconciler/src/ReactFiberWorkLoop.ts
function performUnitOfWork(unitOfWork: Fiber) {
  // 首屏渲染只有当前应用的根结点存在current，其它节点current为null
  const current = unitOfWork.alternate;
  const next = beginWork(current, unitOfWork, subtreeRenderLanes);
  // 属性已经更新到dom上了，memoizedProps更新为pendingProps
  unitOfWork.memoizedProps = unitOfWork.pendingProps;

  // 不存在子fiber节点了，说明节点已经处理完，此时进入completeWork
  if (next == null) {
    completeUnitOfWork(unitOfWork);
  } else {
    workInProgress = next;
  }
}
```

归阶段始于 completeUnitOfWork，它接收一个参数 unitOfWork，值是递阶段处理完成的 fiber 节点。该方法的主要逻辑：

1. 调用 completeWork 方法对递阶段处理完的 fiber 阶段做一些其它处理，主要是DOM节点的处理

2. 当前节点处理完成后会尝试获取其兄弟节点，在我们的 demo 中不存在兄弟节点的清空，因此暂时不需要考虑
3. 兄弟节点不存在则向上处理父 fiber 节点
4. 循环 1-3 步骤直到没有 fiber 节点需要处理

```
// react-reconciler/src/ReactFiberWorkLoop.ts
function completeUnitOfWork(unitOfWork: Fiber) {
  let completedWork = unitOfWork;
  do {
    const current = completedWork.alternate;
    const returnFiber = completedWork.return!;
    completeWork(current, completedWork, subtreeRenderLanes);
    // 处理当前节点的兄弟节点
    const siblingFiber = completedWork.sibling;
    if (siblingFiber !== null) {
      workInProgress = siblingFiber;
      return;
    }
    // returnFiber的子节点已经全部处理完毕，开始处理returnFiber
    completedWork = returnFiber;
    workInProgress = completedWork;
  } while (completedWork !== null);
  // 省略其它代码
}
```

在 completeUnitOfWork 工作步骤中，最关键的还是 completedWork。completedWork 内部会根据 fiber 的类型做不同的逻辑处理。以我们的 demo 为例，在 beginWork 阶段最后处理完的是 span fiber，因此在 completedWork 中会先处理 span fiber 的逻辑。span fiber 属于 HostComponent 类型，该部分在 mount 阶的主要逻辑：

1. 调用 createInstance 方法生成 fiber 对应的 DOM 节点，其内部核心创建方法为 document.createElement
2. 调用 appendAllChildren 将当前 fiber 的child fiber 对应的真实 DOM 添加到自身真实dom 下。这种方式可以将下层的 DOM 汇聚到上层，最终向页面添加 DOM 时只需要 append 最顶层的 DOM 即可

3. 将创建完的 DOM 赋值给 fiber 上的 stateNode 属性
4. 调用 finalizeInitialChildren 初始化 DOM 属性等等
5. bubbleProperties 是优先级冒泡，这里暂不需要了解

```
// demo.jsx
const App = () => <div><span>react</span></div>;

// react-reconciler/src/ReactFiberCompleteWork.ts
export function completeWork(
  current: Fiber | null,
  workInProgress: Fiber,
  renderLanes: Lanes
) {
  const newProps = workInProgress.pendingProps;
  switch (workInProgress.tag) {
    // 函数式组件
    case FunctionComponent: {
      bubbleProperties(workInProgress);
      return null;
    }
    // 当前应用的根结点
    case HostRoot: {
      bubbleProperties(workInProgress);
      return null;
    }
    // 普通元素节点
    case HostComponent: {
      const type = workInProgress.type;
      if (current !== null && workInProgress !== null) {
        // 省略部分代码
      } else {
        // 创建元素
        const instance = createInstance(type, newProps, workInProgress);
        // 在归阶段的时候，子fiber对应的真实dom已经全部创建完毕，此时只需要
        // 将当前fiber节点的child fiber节点对应的真实dom添加到自身真实dom下
```

```
            appendAllChildren(instance, workInProgress);
            // 将stateNode指向当前创建的dom节点
            workInProgress.stateNode = instance;
            // 初始化挂载属性
            finalizeInitialChildren(instance, type, newProps);
          }
          bubbleProperties(workInProgress);
          return null;
        }
        // 省略其它代码
      }
      return null;
    }
```

至此，render 阶段的工作大致告一段落。由于我们的 demo 例子比较简单，所以还有其它情况没有详细说明，包括多子节点、Fragment、class 组件等逻辑的处理，该部分在后续会慢慢补充，下一章将介绍 commit 阶段的流程。