8-6 render 阶段-beginWork

如果以下源码没有特殊标记路径，react/packages/react-reconciler/src/ReactFiberBeginWork.js

本节着重讲初次渲染页面的场景。

DebugReact > src > react > packages > react-reconciler > src > JS ReactFiberBeginWork.js

```
3917  function beginWork(
3918    current: Fiber | null,
3919    workInProgress: Fiber,
3920    renderLanes: Lanes,
3921  ): Fiber | null {
```

1. 初始化

JavaScript

```
didReceiveUpdate = false;
//
workInProgress.lanes = NoLanes;
```

2. 根据组件类型，执行相关 updateXComponent 函数

这里列举函数组件、类组件和 HostRoot 的部分代码：

JavaScript

```
switch (workInProgress.tag) {
    // ...
    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,
      );
    }
    case ClassComponent: {
      const Component = workInProgress.type;
      const unresolvedProps = workInProgress.pendingProps;
      const resolvedProps =
        workInProgress.elementType === Component
          ? unresolvedProps
          : resolveDefaultProps(Component, unresolvedProps);
      return updateClassComponent(
        current,
        workInProgress,
        Component,
        resolvedProps,
        renderLanes,
      );
    }
    case HostRoot:
      return updateHostRoot(current, workInProgress, renderLanes);
```

```
    // ...
}
```

updateHostRoot

1. 更新当前 fiber，比如 props/state 更新，生命周期函数执行、Hooks 函数执行等。
2. 返回一个下一个 fiber。

1. 更新 props、state、updateQuue、transition

JavaScript

```
const nextProps = workInProgress.pendingProps;
  const prevState = workInProgress.memoizedState;
  const prevChildren = prevState.element;
  cloneUpdateQueue(current, workInProgress);
  processUpdateQueue(workInProgress, nextProps, null, renderLanes);

  // processUpdateQueue 阶段可能计算出了新的state
  const nextState: RootState = workInProgress.memoizedState;
  const root: FiberRoot = workInProgress.stateNode;
```

cloneUpdateQueue

把 current 上的 updateQueue 复用到 workInProgress 上一份。

JavaScript

```
export function cloneUpdateQueue<State>(
  current: Fiber,
  workInProgress: Fiber,
): void {
  // Clone the update queue from current. Unless it's already a clone.
  const queue: UpdateQueue<State> = (workInProgress.updateQueue: any);
  const currentQueue: UpdateQueue<State> = (current.updateQueue: any);
  if (queue === currentQueue) {
    const clone: UpdateQueue<State> = {
      baseState: currentQueue.baseState,
      firstBaseUpdate: currentQueue.firstBaseUpdate,
      lastBaseUpdate: currentQueue.lastBaseUpdate,
```

```
      shared: currentQueue.shared,
      callbacks: null,
    };
    workInProgress.updateQueue = clone;
  }
}
```

`processUpdateQueue`

在《update 的数据结构与算法》章节已经讲过，这里不再展开。

2.1 bailout

bailout 阶段，返回 null

不会发生在组件初次渲染阶段，仅仅发生在组件更新阶段。当组件子节点没有发生变化，或者是被手动挡住（如类组件的 shouldComponentUpdate、memo 等），组件子节点不需要协调的时候，如：

DebugReact > src > react > packages > react-reconciler > src > JS ReactFiberBeginWork.js > ...

```
1357  const nextProps = workInProgress.pendingProps;
1358  const prevState = workInProgress.memoizedState;
1359  const prevChildren = prevState.element;
1360  cloneUpdateQueue(current, workInProgress);
1361  processUpdateQueue(workInProgress, nextProps, null, renderLanes);
1362
1363  const nextState: RootState = workInProgress.memoizedState;
1364  const root: FiberRoot = workInProgress.stateNode;
1365  pushRootTransition(workInProgress, root, renderLanes);
1366
1367  if (enableTransitionTracing) {…
1369  }
1370
1371  if (enableCache) {…
1378  }
1379
1380  // This would ideally go inside processUpdateQueue, but because it suspends,…
1383  suspendIfUpdateReadFromEntangledAsyncAction();
1384
1385  // Caution: React DevTools currently depends on this property…
1387  const nextChildren = nextState.element;
1388  if (supportsHydration && prevState.isDehydrated) {…
1463  } else {
1464    // Root is not dehydrated. Either this is a client-only root, or it
1465    // already hydrated.
1466    // ! 2. bailout or 协调子节点
1467    resetHydrationState();
1468    if (nextChildren === prevChildren) {
1469      return bailoutOnAlreadyFinishedWork(current, workInProgress, renderLanes);
1470    }
1471    reconcileChildren(current, workInProgress, nextChildren, renderLanes);
1472  }
1473  // ! 3. 返回子节点
1474  return workInProgress.child;
1475 }
```

`bailoutOnAlreadyFinishedWork`

此函数在组件更新阶段再讲细节。

2.2 协调子节点

```JavaScript
const nextChildren = nextState.element;

if (nextChildren === prevChildren) {
  return bailoutOnAlreadyFinishedWork(current, workInProgress, renderL
}
reconcileChildren(current, workInProgress, nextChildren, renderLanes);
```

bailout:

index.js

```JavaScript
const root = createRoot(document.getElementById("root"), {

root.render(jsx);

setTimeout(() => {
  root.render(jsx);
}, 1000);
```

协调子节点 reconcileChildren

进入协调阶段：

```JavaScript
export function reconcileChildren(
  current: Fiber | null,
  workInProgress: Fiber,
  nextChildren: any,
  renderLanes: Lanes,
) {
  if (current === null) {
    // 组件初次挂载
    workInProgress.child = mountChildFibers(
      workInProgress,
```

```
      null,
      nextChildren,
      renderLanes,
    );
  } else {
    // 组件更新
    workInProgress.child = reconcileChildFibers(
      workInProgress,
      current.child,
      nextChildren,
      renderLanes,
    );
  }
}
```

其实最终执行的函数是同一个，都是 createChildReconciler，而 createChildReconciler 是个 wrapper function。

JavaScript

```
export const reconcileChildFibers: ChildReconciler =
  createChildReconciler(true);
export const mountChildFibers: ChildReconciler = createChildReconciler
```

这里简单介绍下页面初次渲染时候的协调，协调的难点在更新，详情在协调章节继续查看。

单个子节点

初次渲染页面的时候，`reconcileSingleElement` 返回新创建的 fiber，`placeSingleChild` 也只是返回这个 fiber。

JavaScript

```
return placeSingleChild(
  reconcileSingleElement(
    returnFiber,
    currentFirstChild,
    newChild,
    lanes,
  ),
);
```

`reconcileSingleElement`

JavaScript

```
function reconcileSingleElement(
    returnFiber: Fiber,
    currentFirstChild: Fiber | null,
    element: ReactElement,
    lanes: Lanes,
    debugInfo: ReactDebugInfo | null,
  ): Fiber {
    // 略...
    if (element.type === REACT_FRAGMENT_TYPE) {
      // 略...
    } else {
      const created = createFiberFromElement(element, returnFiber.mode
      coerceRef(returnFiber, currentFirstChild, created, element);
      created.return = returnFiber;
      return created;
    }
  }
```

多个子节点

当子节点是多个的时候，需要遍历生成 fiber，并把它们链接成单链表结构。

JavaScript

```
function reconcileChildrenArray(
  returnFiber: Fiber,
  currentFirstChild: Fiber | null,
  newChildren: Array<any>,
  lanes: Lanes,
  debugInfo: ReactDebugInfo | null,
): Fiber | null {
    let resultingFirstChild: Fiber | null = null;
    let previousNewFiber: Fiber | null = null;

    let oldFiber = currentFirstChild;
    let lastPlacedIndex = 0;
    let newIdx = 0;
    // 略...
    if (oldFiber === null) {
```

```
    for (; newIdx < newChildren.length; newIdx++) {
      const newFiber = createChild(
        returnFiber,
        newChildren[newIdx],
        lanes,
        debugInfo,
      );
      if (newFiber === null) {
        continue;
      }
      lastPlacedIndex = placeChild(newFiber, lastPlacedIndex, newIdx);
      if (previousNewFiber === null) {
        resultingFirstChild = newFiber;
      } else {
        previousNewFiber.sibling = newFiber;
      }
      previousNewFiber = newFiber;
    }
    return resultingFirstChild;
  }
  // 略...
}
```

创建子 Fiber

JavaScript

```
function createChild(
   returnFiber: Fiber,
   newChild: any,
   lanes: Lanes,
 ): Fiber | null {
   if (
     (typeof newChild === 'string' && newChild !== '') ||
     typeof newChild === 'number'
   ) {
     const created = createFiberFromText(
       '' + newChild,
       returnFiber.mode,
       lanes,
     );
     created.return = returnFiber;
```

```
      return created;
    }

    if (typeof newChild === 'object' && newChild !== null) {
      switch (newChild.$$typeof) {
        case REACT_ELEMENT_TYPE: {
          const created = createFiberFromElement(
            newChild,
            returnFiber.mode,
            lanes,
          );
          coerceRef(returnFiber, null, created, newChild);
          created.return = returnFiber;
          if (__DEV__) {
            created._debugInfo = mergeDebugInfo(debugInfo, newChild._d
          }
          return created;
        }
        // 略 ...
      }

      if (isArray(newChild) || getIteratorFn(newChild)) {
        const created = createFiberFromFragment(
          newChild,
          returnFiber.mode,
          lanes,
          null,
        );
        created.return = returnFiber;
        return created;
      }

    // 略 ...
    }
    return null;
  }
```

3. 返回子节点

JavaScript

```
return workInProgress.child;
```