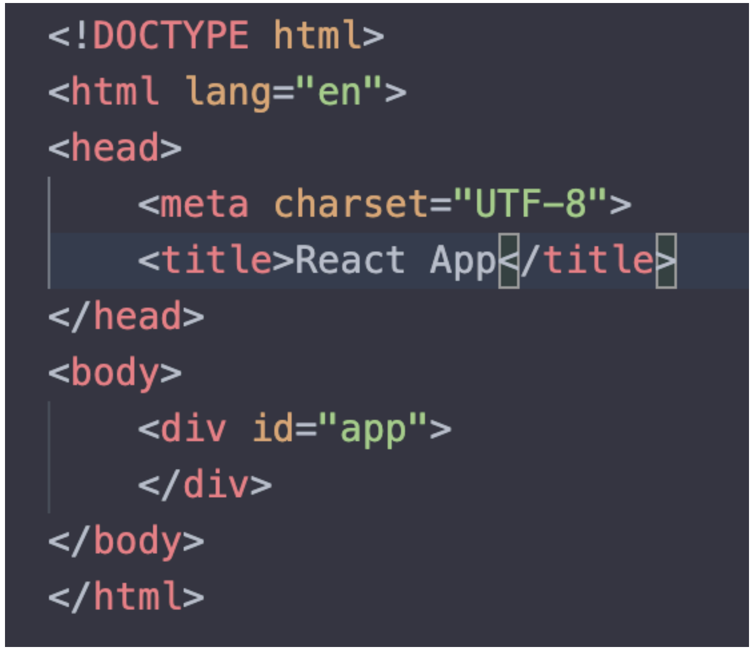
React首次渲染

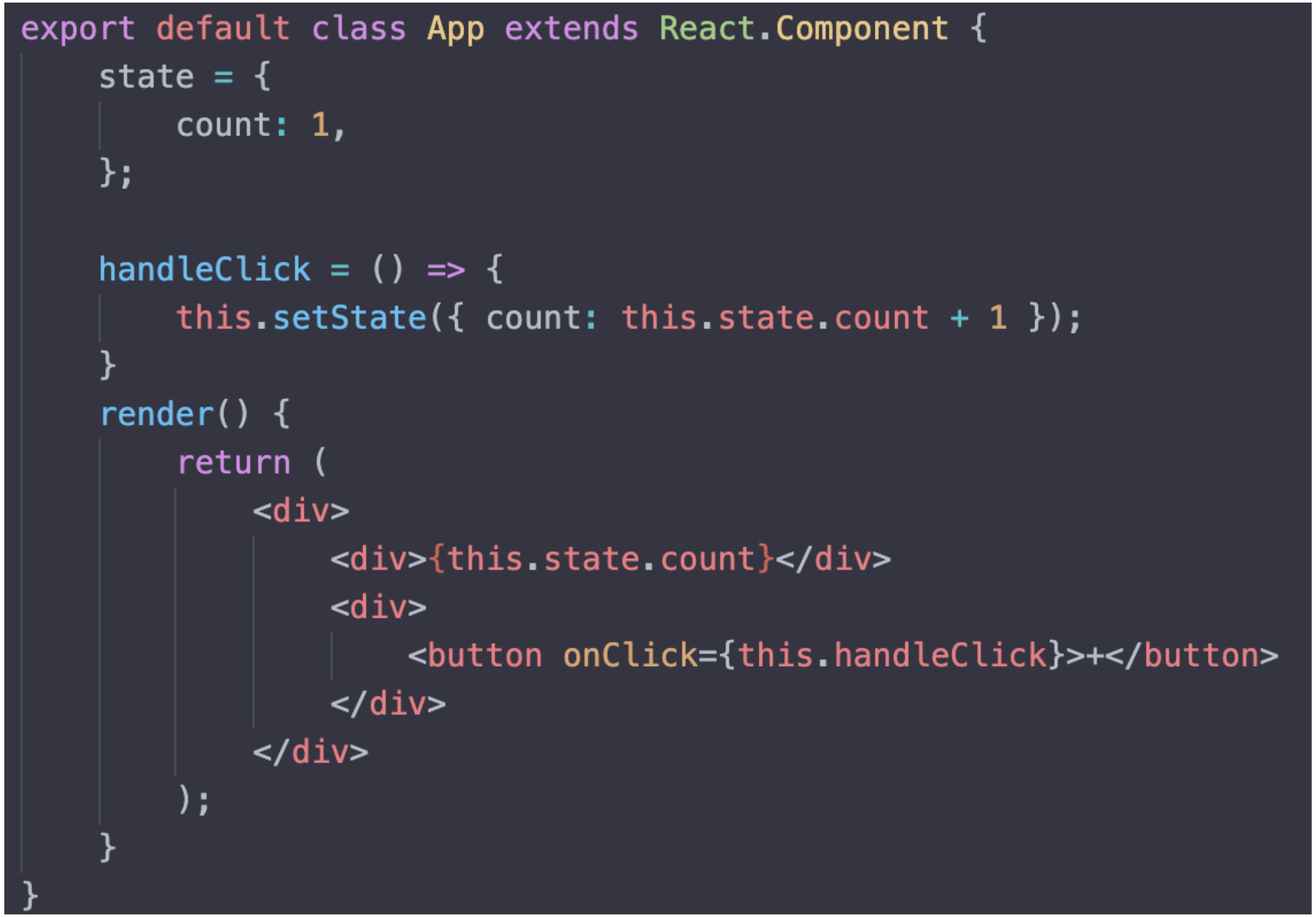
V16.12.0

同步渲染

以一个例子开始

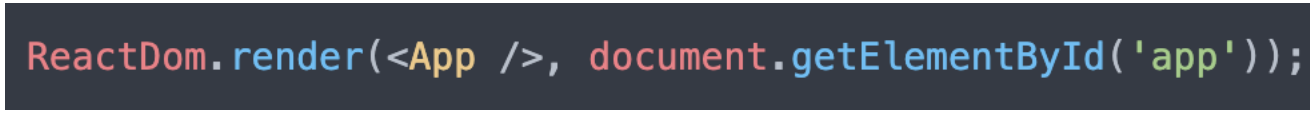


```html
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>React App</title>
</head>
<body>
    <div id="app">
    </div>
</body>
</html>
```



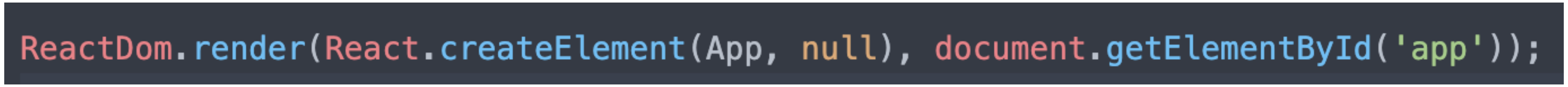
```jsx
export default class App extends React.Component {
    state = {
        count: 1,
    };

    handleClick = () => {
        this.setState({ count: this.state.count + 1 });
    }
    render() {
        return (
            <div>
                <div>{this.state.count}</div>
                <div>
                    <button onClick={this.handleClick}>+</button>
                </div>
            </div>
        );
    }
}
```

element转换



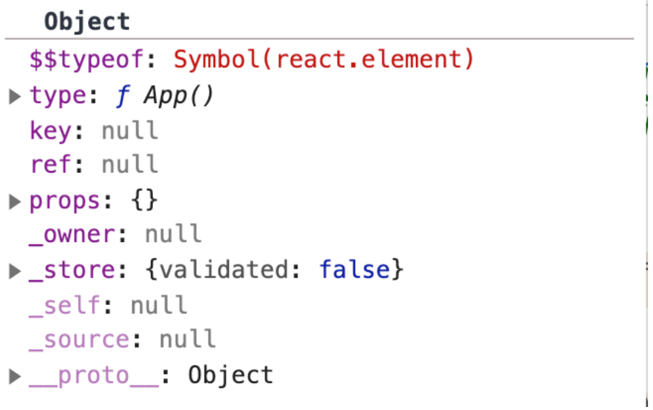
```
ReactDom.render(<App />, document.getElementById('app'));
```

babel转换



```
ReactDom.render(React.createElement(App, null), document.getElementById('app'));
```

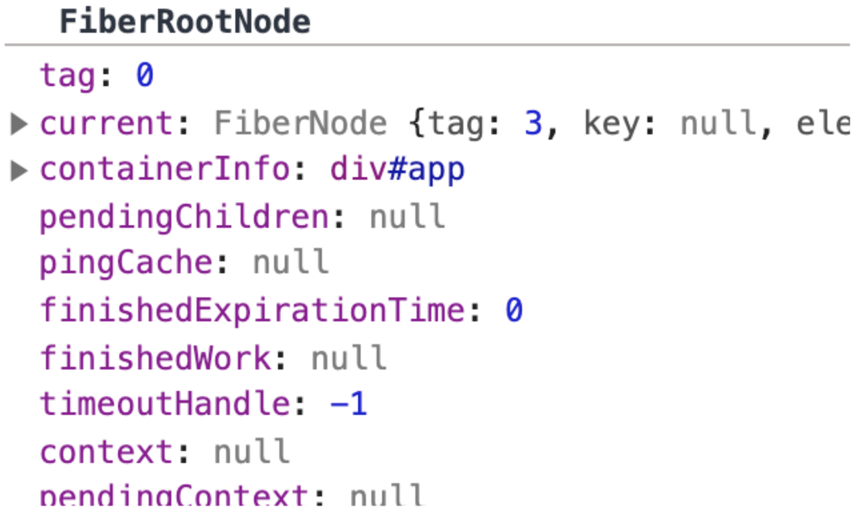
element信息



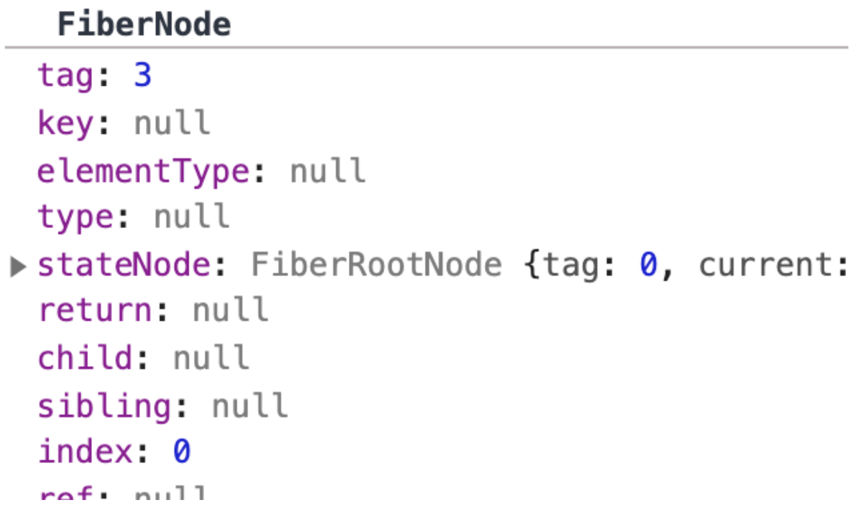
```
  Object
  $$typeof: Symbol(react.element)
▶ type: ƒ App()
  key: null
  ref: null
▶ props: {}
  _owner: null
▶ _store: {validated: false}
  _self: null
  _source: null
▶ __proto__: Object
```

fiberRoot和HostRoot创建

- 清空挂载dom下所有子节点
- 创建fiberRoot和hostRoot，hostRoot放置在fiberRoot.current下



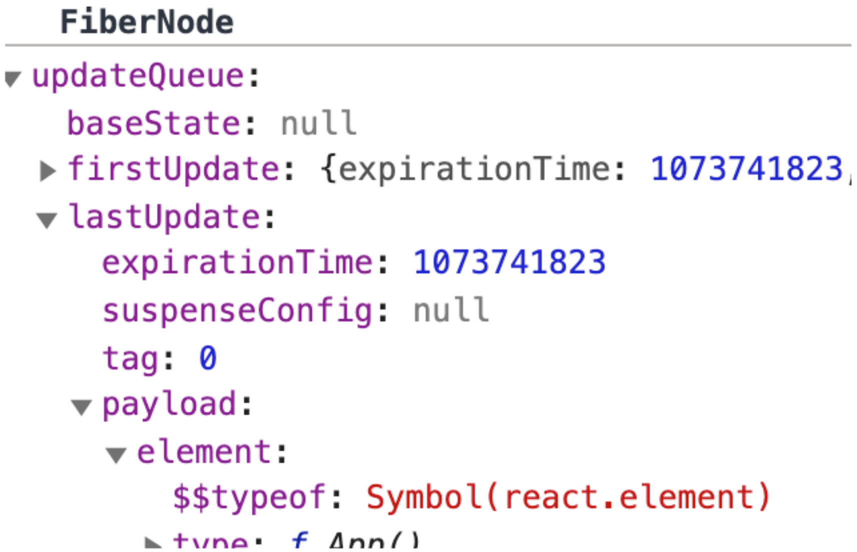
```
 FiberRootNode
 tag: 0
▶current: FiberNode {tag: 3, key: null, ele
▶containerInfo: div#app
 pendingChildren: null
 pingCache: null
 finishedExpirationTime: 0
 finishedWork: null
 timeoutHandle: -1
 context: null
 pendingContext: null
```



```
 FiberNode
 tag: 3
 key: null
 elementType: null
 type: null
▶stateNode: FiberRootNode {tag: 0, current:
 return: null
 child: null
 sibling: null
 index: 0
 ref: null
```

更新hostRoot

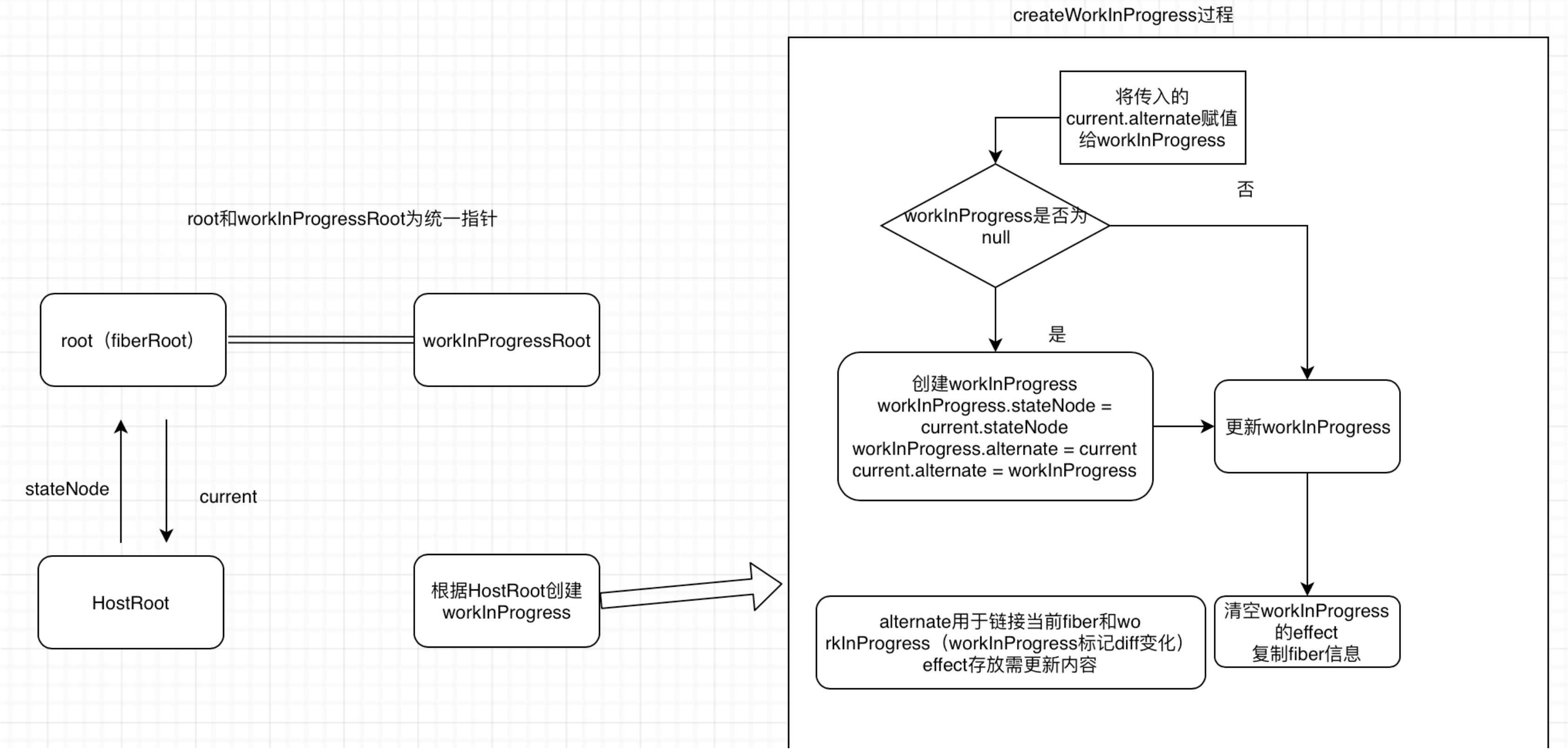
- 创建update， 将element放置update.payload
- 创建updateQueue， 添加update， 更新hostRoot.updateQueue



```
 FiberNode
▼ updateQueue:
    baseState: null
  ▶ firstUpdate: {expirationTime: 1073741823,
  ▼ lastUpdate:
      expirationTime: 1073741823
      suspenseConfig: null
      tag: 0
    ▼ payload:
      ▼ element:
          $$typeof: Symbol(react.element)
        ▶ type: f App()
```

performSyncWorkOnRoot

- 以hostRoot克隆workInProgress（更新树）



root和workInProgressRoot为统一指针

root（fiberRoot）

workInProgressRoot

stateNode

current

HostRoot

根据HostRoot创建 workInProgress

createWorkInProgress过程

将传入的 current.alternate赋值 给workInProgress

workInProgress是否为 null

否

是

创建workInProgress
workInProgress.stateNode = current.stateNode
workInProgress.alternate = current
current.alternate = workInProgress

更新workInProgress

alternate用于链接当前fiber和workInProgress（workInProgress标记diff变化）effect存放需更新内容

清空workInProgress 的effect 复制fiber信息

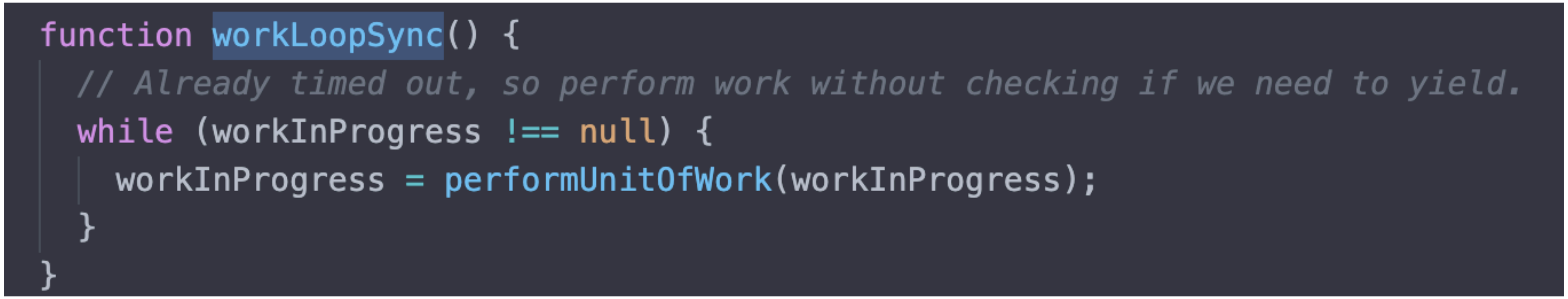
current.alternate代表更新树
workInProgress.alternate代表旧树

React使用双树作为缓冲池，快速比较新树和旧树

注：只要生成了双树，缓冲池则已建立，后续只更新属性，不再创建

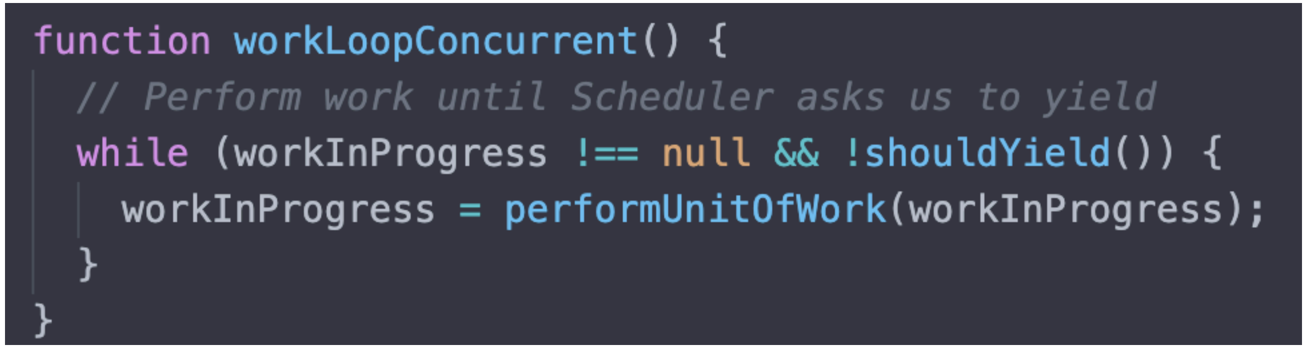
workLoopSync

同步相对比较简单，处理完所有的workInProgress才会结束



```
function workLoopSync() {
  // Already timed out, so perform work without checking if we need to yield.
  while (workInProgress !== null) {
    workInProgress = performUnitOfWork(workInProgress);
  }
}
```

异步会检验是否需要中断，不需要才会处理下一个workInProgress（异步渲染以后再分享）

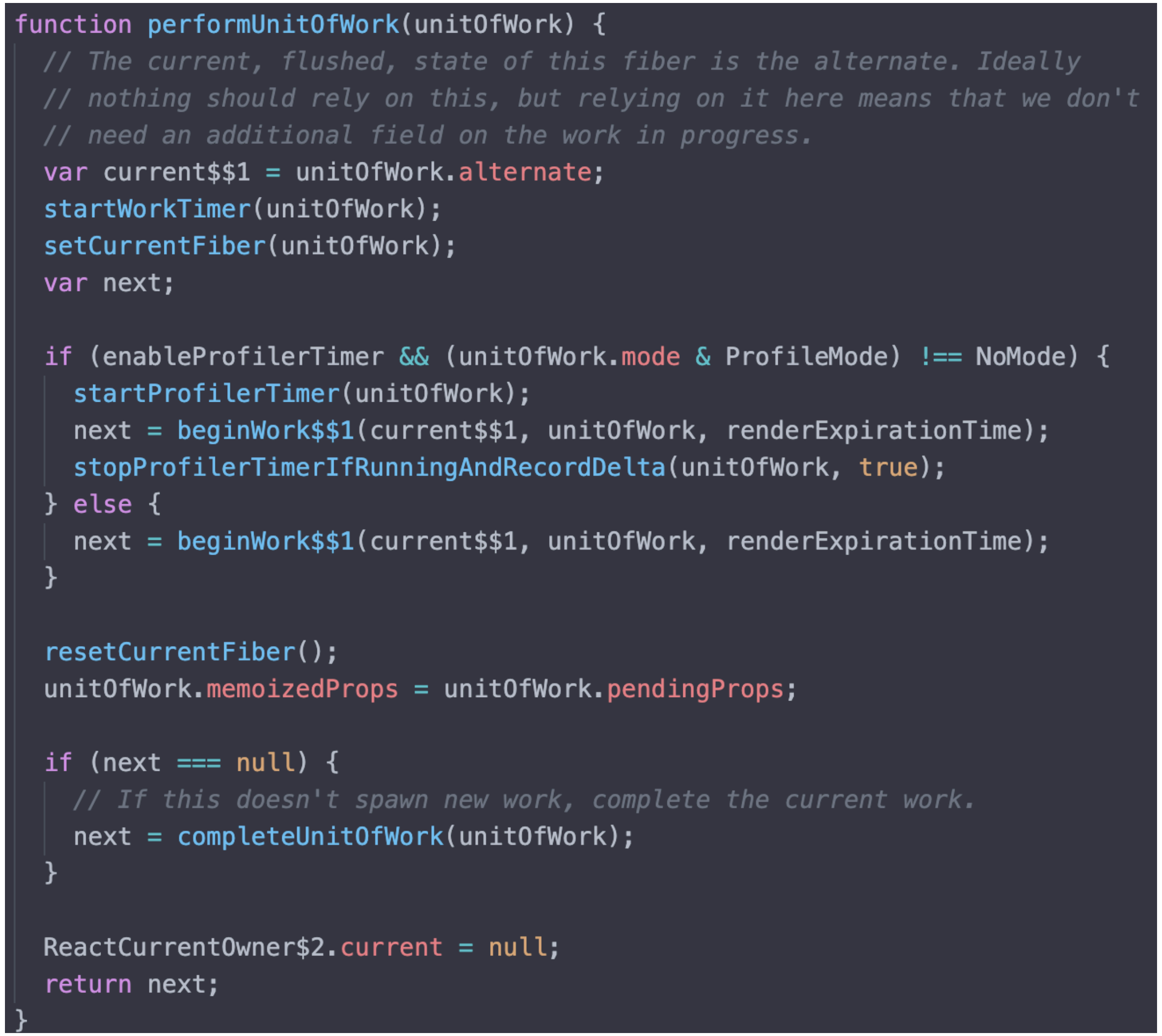


```
function workLoopConcurrent() {
  // Perform work until Scheduler asks us to yield
  while (workInProgress !== null && !shouldYield()) {
    workInProgress = performUnitOfWork(workInProgress);
  }
}
```

performUnitOfWork

- 调用beginWork获取next（workInProgress.child）
- 如果next为空，调用completeUnitOfWork获取next，

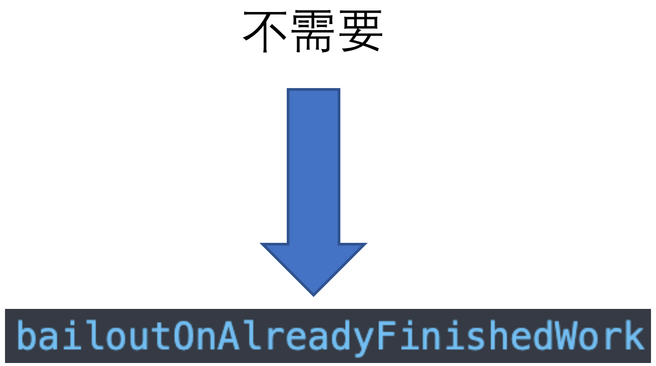
返回当前workInProgress.sibiling（兄弟节点），
没有则找父fiber.sibiling，……，直到父fiber不存在。
同时创建或更新dom信息，然后将这些effect信息
逐级传给父fiber。



```
function performUnitOfWork(unitOfWork) {
  // The current, flushed, state of this fiber is the alternate. Ideally
  // nothing should rely on this, but relying on it here means that we don't
  // need an additional field on the work in progress.
  var current$$1 = unitOfWork.alternate;
  startWorkTimer(unitOfWork);
  setCurrentFiber(unitOfWork);
  var next;

  if (enableProfilerTimer && (unitOfWork.mode & ProfileMode) !== NoMode) {
    startProfilerTimer(unitOfWork);
    next = beginWork$$1(current$$1, unitOfWork, renderExpirationTime);
    stopProfilerTimerIfRunningAndRecordDelta(unitOfWork, true);
  } else {
    next = beginWork$$1(current$$1, unitOfWork, renderExpirationTime);
  }

  resetCurrentFiber();
  unitOfWork.memoizedProps = unitOfWork.pendingProps;

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

  ReactCurrentOwner$2.current = null;
  return next;
}
```

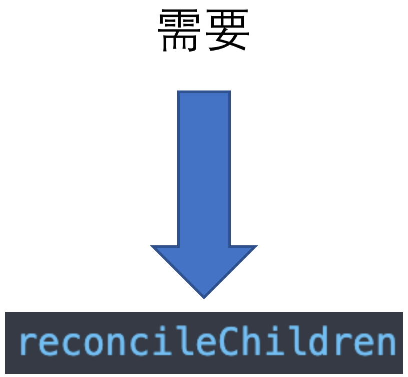
beginWork

- 根据workInProgress.tag选择不同的处理方式获取下一个workInProgress（child）
- hostRoot（举例）：处理更新队列，计算newState，比较prevState.element和newState.element
- 不同处理方法大致都差不多，比较props、state、context变化或者forceUpdate来决定当前workInProgress是否需要更新，需要更新的情况下获取nextChildren（比如class组件调render）。



不需要

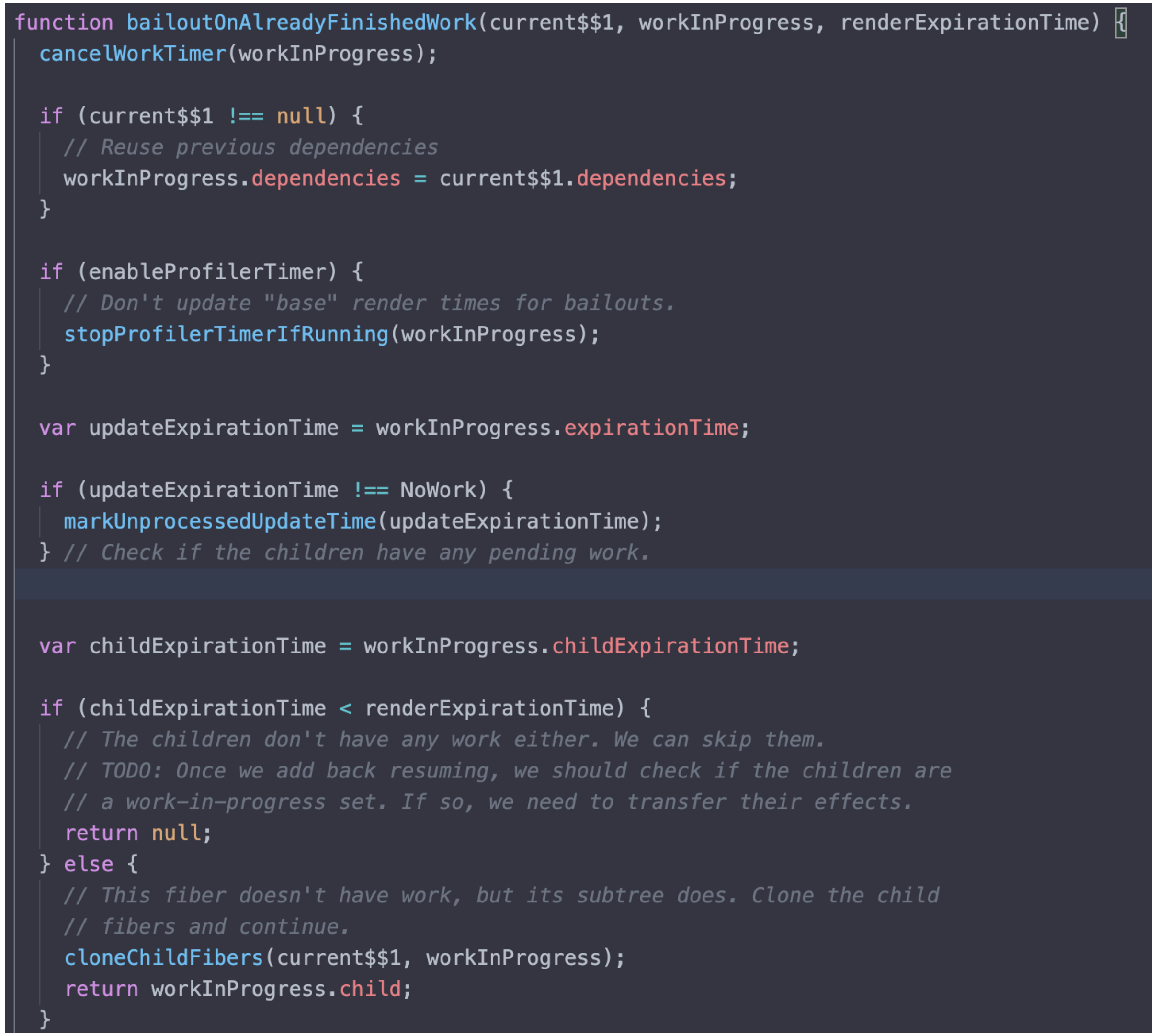
```
bailoutOnAlreadyFinishedWork
```



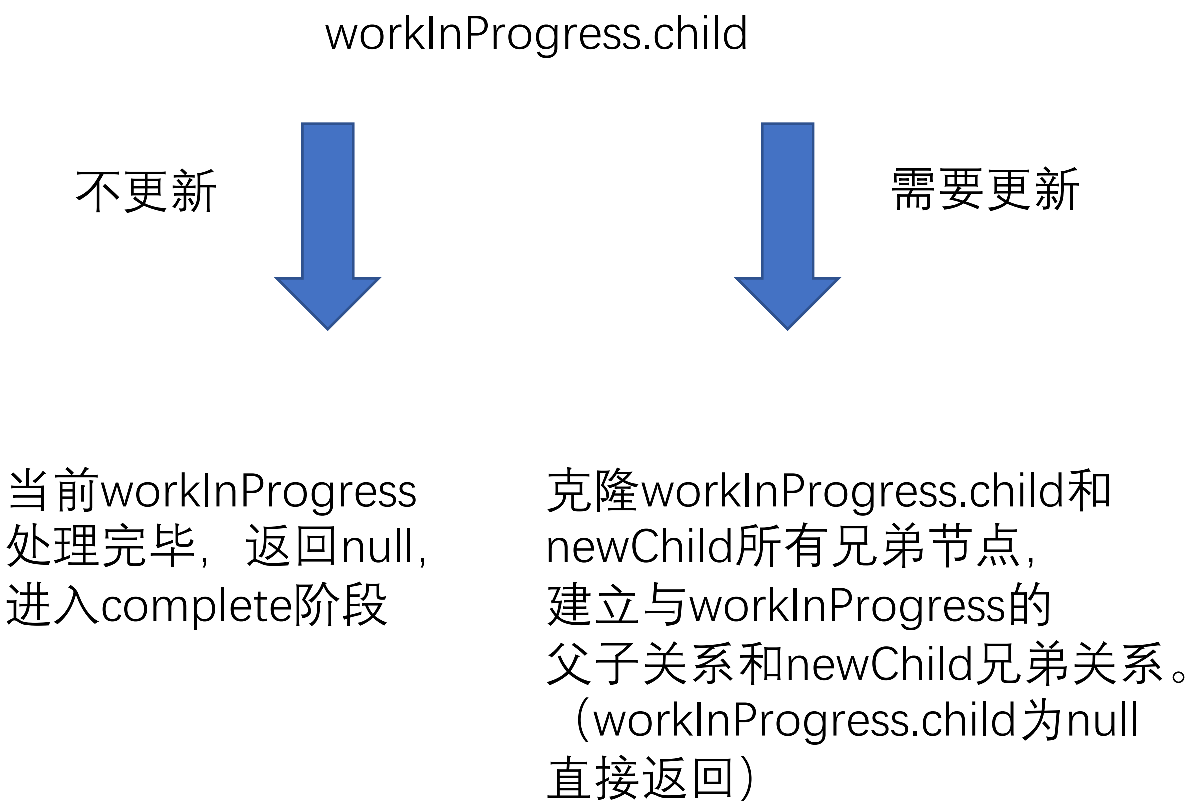
需要

```
reconcileChildren
```

bailoutOnAlreadyFinishedWork



```
function bailoutOnAlreadyFinishedWork(current$$1, workInProgress, renderExpirationTime) {
  cancelWorkTimer(workInProgress);

  if (current$$1 !== null) {
    // Reuse previous dependencies
    workInProgress.dependencies = current$$1.dependencies;
  }

  if (enableProfilerTimer) {
    // Don't update "base" render times for bailouts.
    stopProfilerTimerIfRunning(workInProgress);
  }

  var updateExpirationTime = workInProgress.expirationTime;

  if (updateExpirationTime !== NoWork) {
    markUnprocessedUpdateTime(updateExpirationTime);
  } // Check if the children have any pending work.


  var childExpirationTime = workInProgress.childExpirationTime;

  if (childExpirationTime < renderExpirationTime) {
    // The children don't have any work either. We can skip them.
    // TODO: Once we add back resuming, we should check if the children are
    // a work-in-progress set. If so, we need to transfer their effects.
    return null;
  } else {
    // This fiber doesn't have work, but its subtree does. Clone the child
    // fibers and continue.
    cloneChildFibers(current$$1, workInProgress);
    return workInProgress.child;
  }
```



workInProgress.child

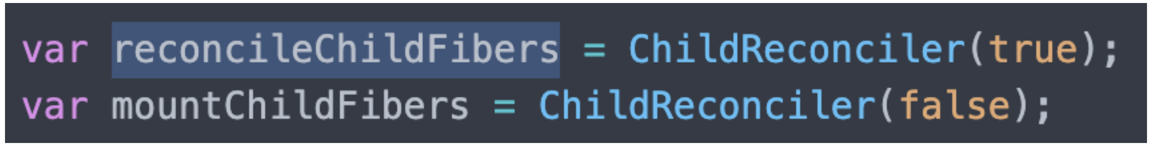
不更新

需要更新

当前workInProgress
处理完毕，返回null，
进入complete阶段

克隆workInProgress.child和
newChild所有兄弟节点，
建立与workInProgress的
父子关系和newChild兄弟关系。
（workInProgress.child为null
直接返回）

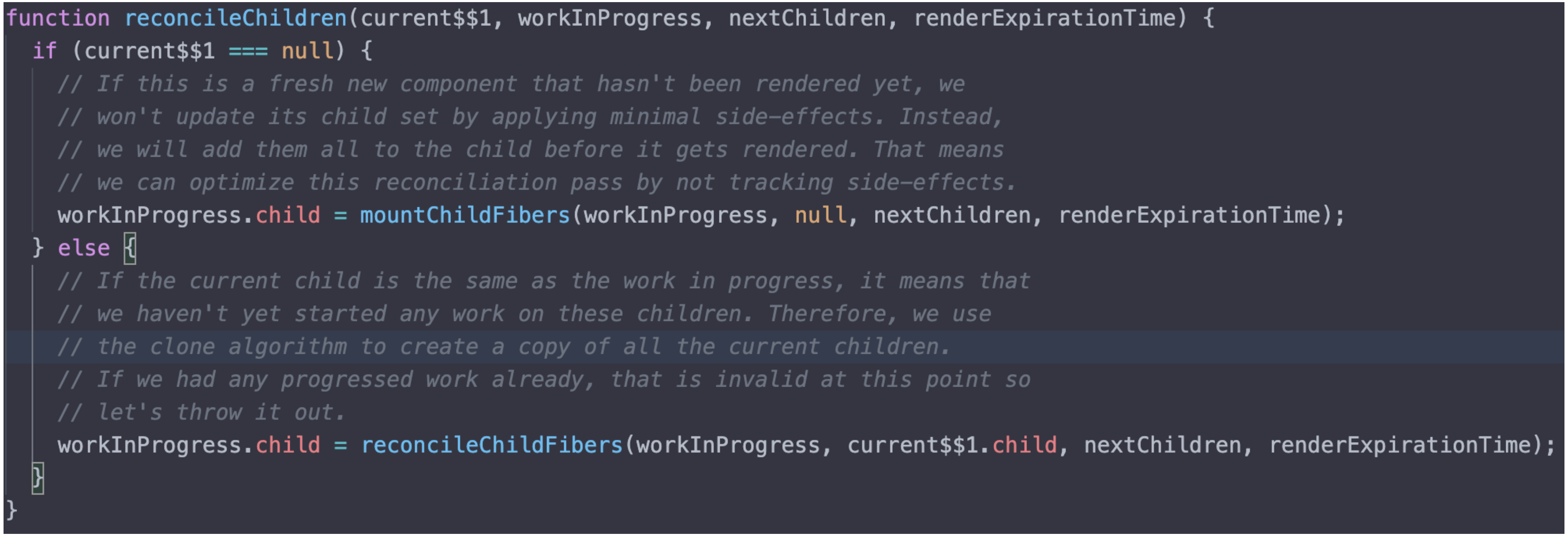
reconcileChildren



```
var reconcileChildFibers = ChildReconciler(true);
var mountChildFibers = ChildReconciler(false);
```

计算新的childFiber，赋值给workInProgress.child

mountChildFibers不需要更新workInProgress.effect链表，reconcileChildFibers需要更新



```
function reconcileChildren(current$$1, workInProgress, nextChildren, renderExpirationTime) {
  if (current$$1 === null) {
    // If this is a fresh new component that hasn't been rendered yet, we
    // won't update its child set by applying minimal side-effects. Instead,
    // we will add them all to the child before it gets rendered. That means
    // we can optimize this reconciliation pass by not tracking side-effects.
    workInProgress.child = mountChildFibers(workInProgress, null, nextChildren, renderExpirationTime);
  } else {
    // If the current child is the same as the work in progress, it means that
    // we haven't yet started any work on these children. Therefore, we use
    // the clone algorithm to create a copy of all the current children.
    // If we had any progressed work already, that is invalid at this point so
    // let's throw it out.
    workInProgress.child = reconcileChildFibers(workInProgress, current$$1.child, nextChildren, renderExpirationTime);
  }
}
```

reconcileChildFibers

- 依据nextChildren调用不同处理方法，按需更新child（旧fiber）（创建、更新和删除，更新childFiber.effectTag类型），返回新的childFiber。
- 创建: 旧child为null，创建fiber，与workInProgress建立父关系，更新fiber.effectTag为Placement，更新chlid为兄弟节点，继续key判断处理直至child为null。
- 删除: 比较新旧child的key，不同直接删除，将child添加至父fiber的effect链表中，更新child.effectTag为Deletion（详见deleteChild），更新chlid为兄弟节点，继续key判断处理直至child为null。
- 更新: 新旧key相同，更新child所有兄弟节点effectTag为Deletion，并添加至父fiber的effect链表中，基于child和nextChildren克隆childFiber，更新childFiber.index = 0；childFiber.sibling = null；（**数组依据index和key达到按需更新，sibling同时更新**）

Ref按需更新至childFiber.ref

completeUnitOfWork （获取next）

- completeWork: 创建dom和更新dom属性，赋值workInProgress.stateNode，更新workInProgress.effectTag |= Update。（|或运算）

更新dom属性时，会进行事件注册，同类型事件只会注册一次（后续分享）

注: Suspense组件会等待子组件加载完后re-render，所以直接返回workInProgress。（搭配lazy使用）

- next为null，添加effect链表至父fiber的effect链表中。workInProgress.sibling不为null，返回兄弟节点。否则更新workInProgress为父fiber，继续completeWork获取next，直到父节点为null（所有节点处理完毕，即顶层workInProgress）。

注: 如果组件下所有workInProgress都complete，则会添加当前workInProgress到父fiber的effect链表中。

Commit阶段

- 将更新树（fiberRoot.current.alternate）赋值给fiberRoot.finishedWork，进入commitRoot。

- 取出finishedWork，清空fiberRoot.finishedWork；

- commitBeforeMutationEffects: 循环遍历finishedWork的effect链表，对有Snapshot的effect执行commitBeforeMutationLifeCycles（参数当前effect.alternate作为current（提供prevProps和prevState），effect为finishedWork）。

class组件取fiber.stateNode（instance）执行instance.getSnapshotBeforeUpdate（DOM信息还未提交，让组件在发生更改之前获取dom信息）

function组件执行commitHookEffectList（ UnmountSnapshot, NoHookEffect, finishedWork）触发hooks上相关effect API（后续分享）

commitMutationEffects

- 循环遍历effect链表，提交dom更新到挂载dom节点（替换、更新、删除），更新effectTag
- 更新完后更新fiberRoot.current = finishedWork

commitLayoutEffects

- 循环遍历effect链表，对有update的effect执行commitLifeCycles。
- Class组件执行componentDidMount
- Function组件执行commitHookEffectList（ UnmountLayout, MountLayout, finishedWork）
- 对于有Ref的effect，关联ref

commit结尾

- 循环更新树effect链表，设置effect.nextEffect = null。

谢谢