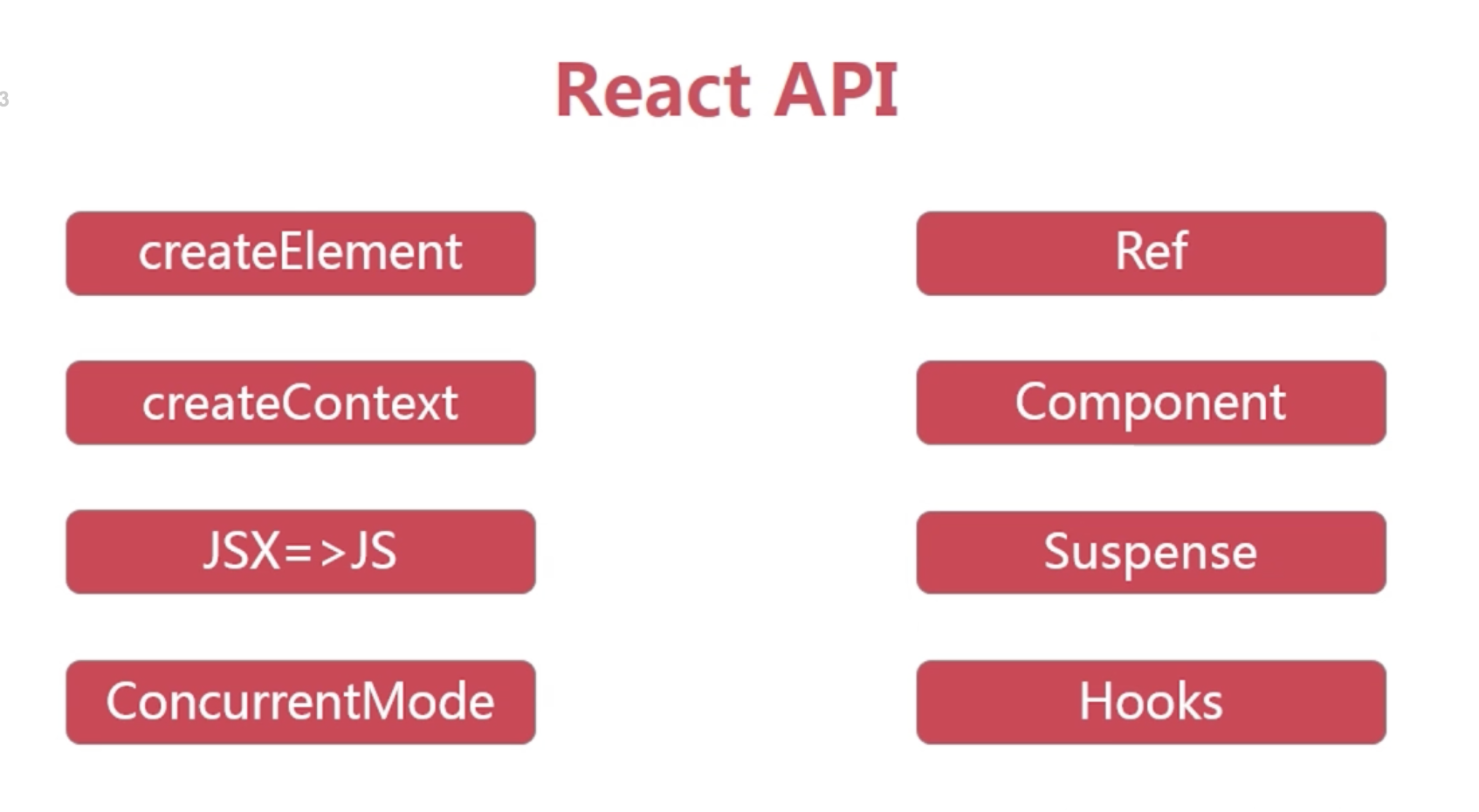
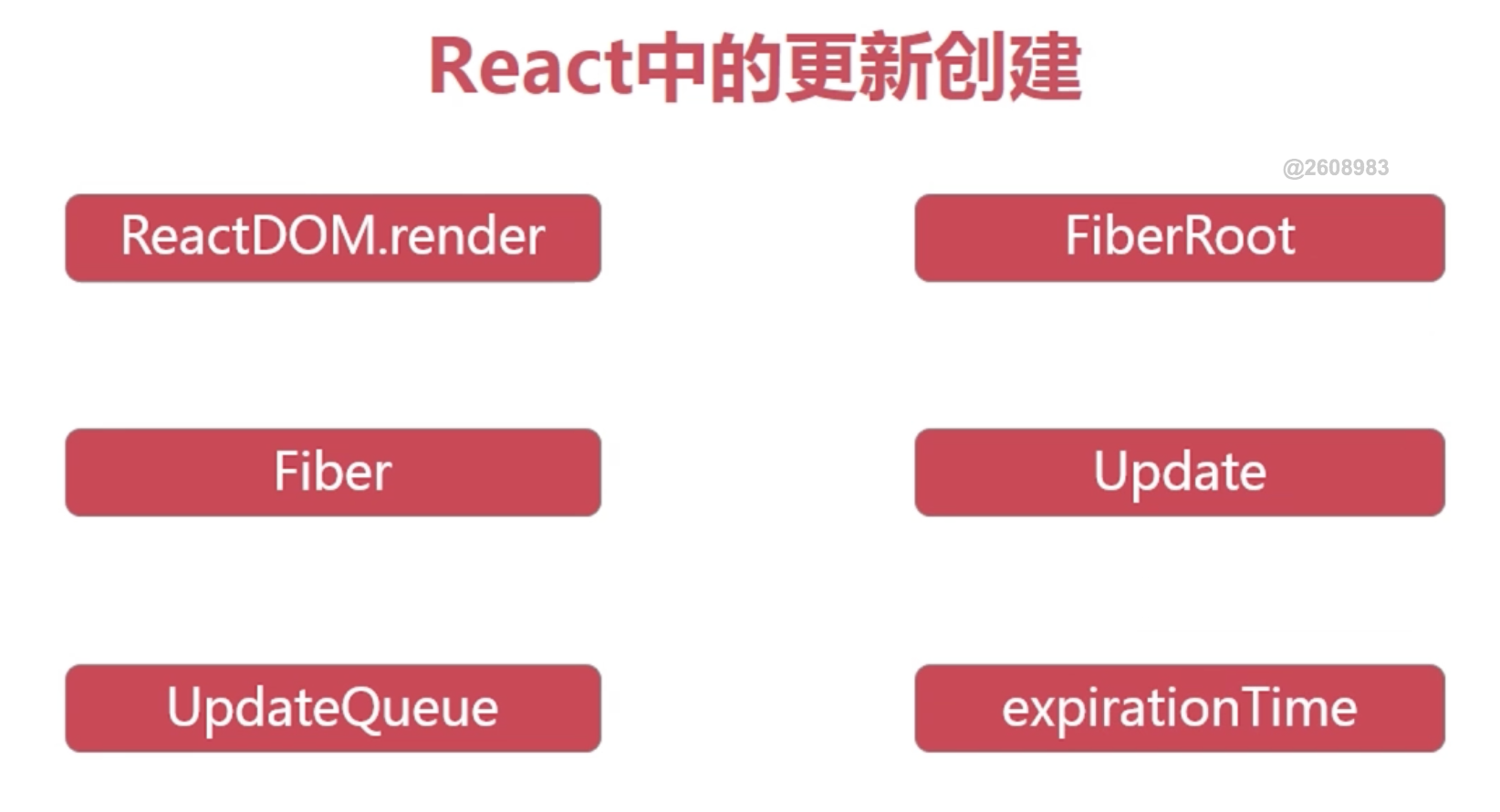
React源码解析

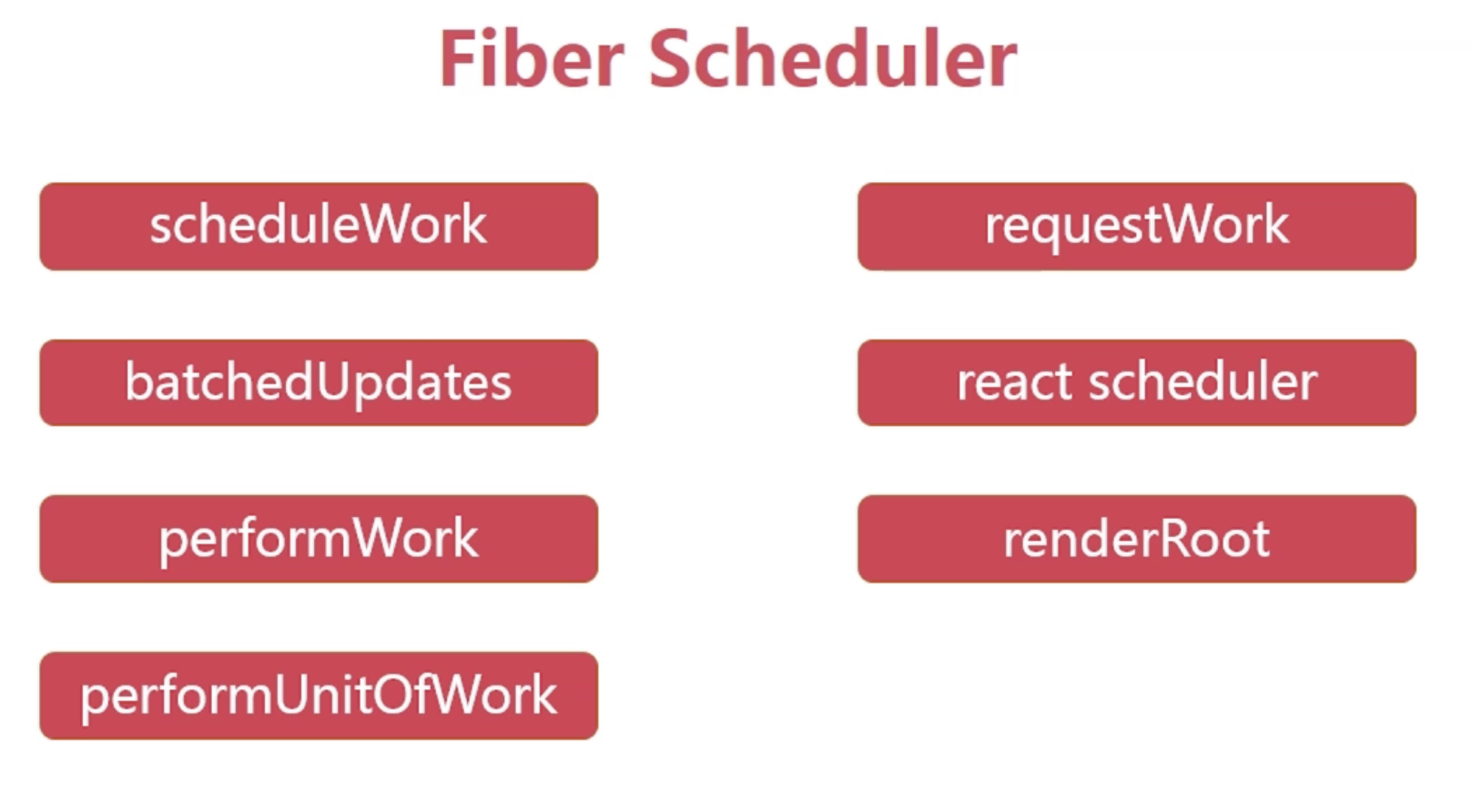
# 课程导学

UI =fn(x):通过state映射UI方式，屏蔽了DOM操作。

API设计非常的纯粹，核心API就是setState,其余所有内容都是围绕着组件化设计，没有双向绑定。一切基于组件。







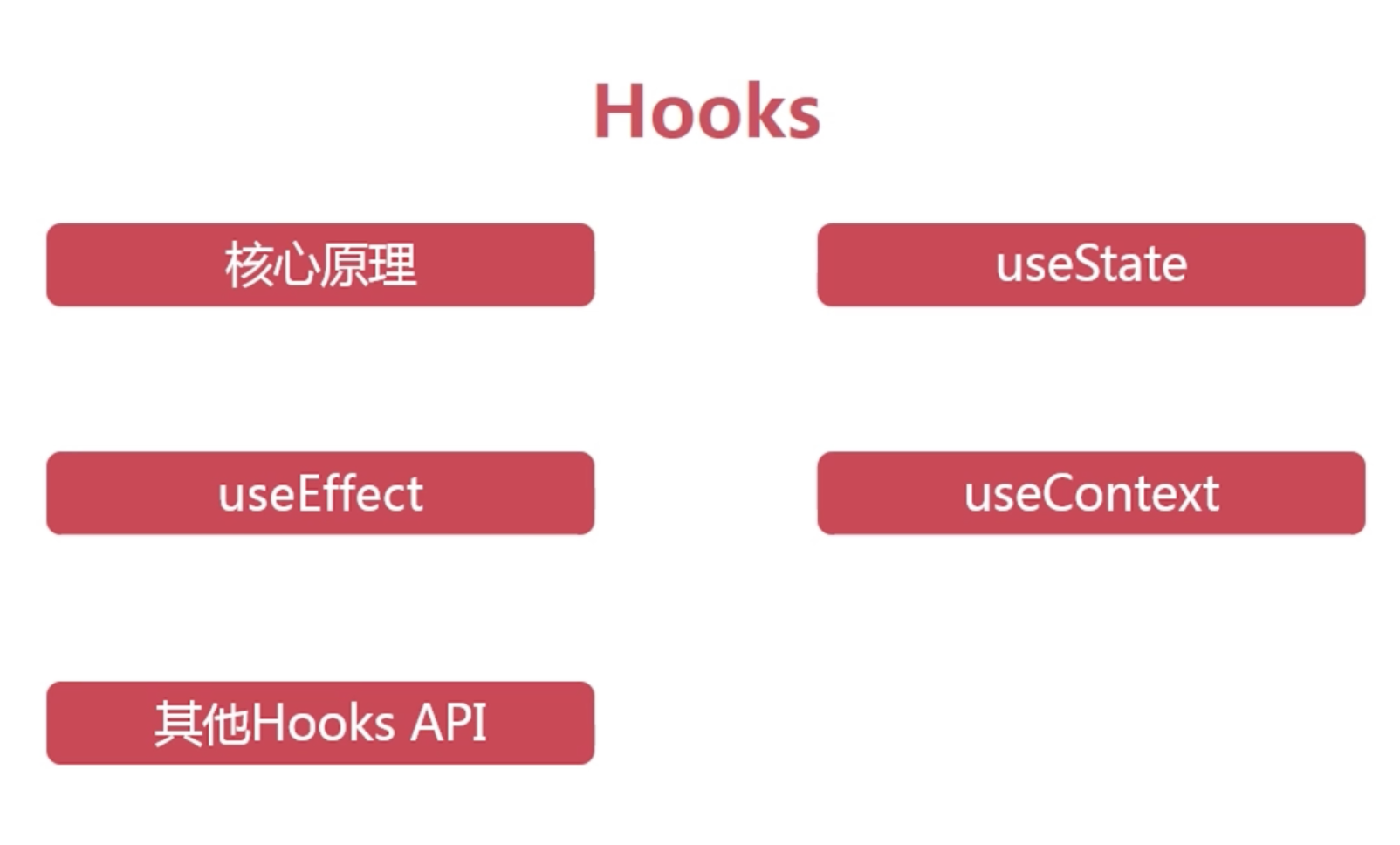












你需要：

足够的耐心

思考再思考

善于提问和记笔记

# 学习源码的执行

* 保持足够的耐心，不要操之过急

一章一章看，一章完全看懂了再看下一章，没有懂继续看这一章。

* 做好笔记

把课程里面老师讲的思想，react源码设计的思路，老师每一章讲的话，用word进行记录下来，源码具体的实现过程，用xmind进行记录下来（记录更多的文字），特别是为什么这么写，有什么判断条件，怎么去实现这些思想，要记录下来。



# 基础知识React API一览

createRef & ref

# react中的更新

1. 总体流程概况react-dom-render

* 创建更新的方式：

ReactDOM.render || hydrate

setState

forceUpdate

* 创建ReactRoot

先要创建ReactRoot，这是一个包含react这整个应用的最顶点的一个对象

* 创建FiberRoot和RootFiber

FiberRoot创建的过程中，会初始化一个Fiber对象。

* 创建更新

创建更新，就会进入更新调度的阶段，进入调度之后，所有的更新都由调度器来管理，跟哪个接口创建更新已经没有关系了。创建一个expirationTime，创建update对象，把update加入到enqueueUpdate中,把这个更新对象放到root对象，进入一个更新流程reactSchedule

1. react-fiber-root

ReactDOM.render的过程中，创建了一个reactRoot的对象，这个对象最主要的任务就是创建了FiberRoot对象。

FiberRoot

* 整个应用的起点
* 包含应用挂载的目标节点
* 记录整个应用过程的各种信息

1. react-fiber

fiber：react16之后的核心；

* 每一个ReactElement对应一个Fiber对象
* 记录节点的各种状态

class Component里面的state和props是记录在Fiber对象上面的，在Fiber更新之后，才会更新到class Component上面的this.state里面和this.props里面。让function Component实现hooks的可能

* 串联整个应用形成树结构

在reactElement当中，通过props.children把我们整个应用串联起来，每个节点通过这个属性拿到子节点。在Fiber里面，也有这样的一个能力来串联整个树形结构。

stateNode:节点对应的实际的实例，class Component对应的实例，DOM组件，对应的DOM的实例。

1. react-update-and-updateQueue

什么是update

* 用来记录组件状态的改变
* 存放于UpdateQueue中

存放update，是一个单向链表的数据结构，一次整体的更新过程当中，可能在这个queue里面存放多个update，在这一次更新过程当中，会根据这些update一个实际的结果，算出一个最终的一个新的state的结果。

* 多个Update可以同时存在

一个事件里面，调用了三次setState，这三次setState产生3个update对象，他们并不会一次setState就更新一个应用，而会等三个SetState执行完，三个update创建完，放倒updateQueue里面，然后再进行一个更新的一个操作。

1. exportation-time
2. different-expiration-time

* Sync模式：

优先级最高，不能被打断

* 异步模式：

会进行调度，会有一系列复杂的调度在里面，会有复杂的操作在里面，可能会被中断，会有一个计算出来的过期时间，

* 指定context

1. react-setState-forceUpdate

* 给节点的Fiber创建更新

ReactDOM.render是创建的Update是放在root上面的，是一个整体的渲染，执行ReactDOM.renders时候，没有任何应用产生，是一个初始化的渲染。setState是针对某一个class Componnent来说的。

* 更新的类型不同

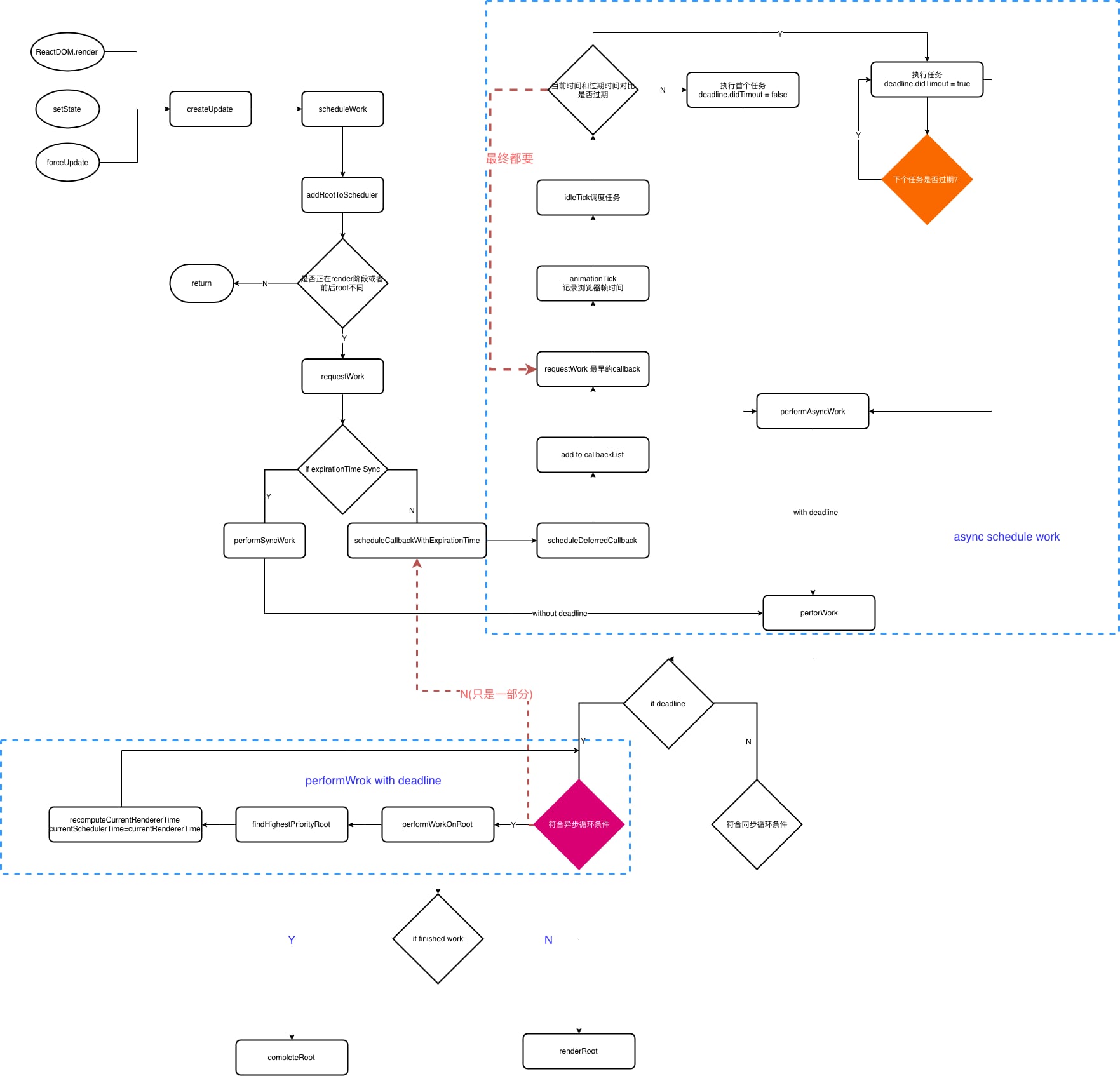
setState和forceUpdate类型不同

# FiberScheduler

1. 总体流程概况

react16之后的reconciler(调和器)我们叫他fiber-reconciler，核心是fiber，fiber的结构帮我们react一整个树的应用的更新的流程能够拆成每一个fiber对象为单元的一个更新的流程，这种单元的形式把更新拆分出来之后，我们可以给每个不同的任务提供一个优先级，以及我们在更新的过程中我们可以中断，我们可以记录我们更新到了那一个单元，中断了之后，可以过一会儿回过头来，继续从这一个单元开始继续之前没有做完的更新，而react16之前，setState产生的更新必须从头到问更新完，然后再执行之后的代码，如果我们的应用树节点非常的多，那么整个更新会导致占用的JS运行时间非常的多，让页面的其他操作进入一个停滞的状态，比如动画的刷新，用户的交互非常卡顿的感觉。react16之后整体的更新流程完全不一样。

* Scheduler的整体流程概览



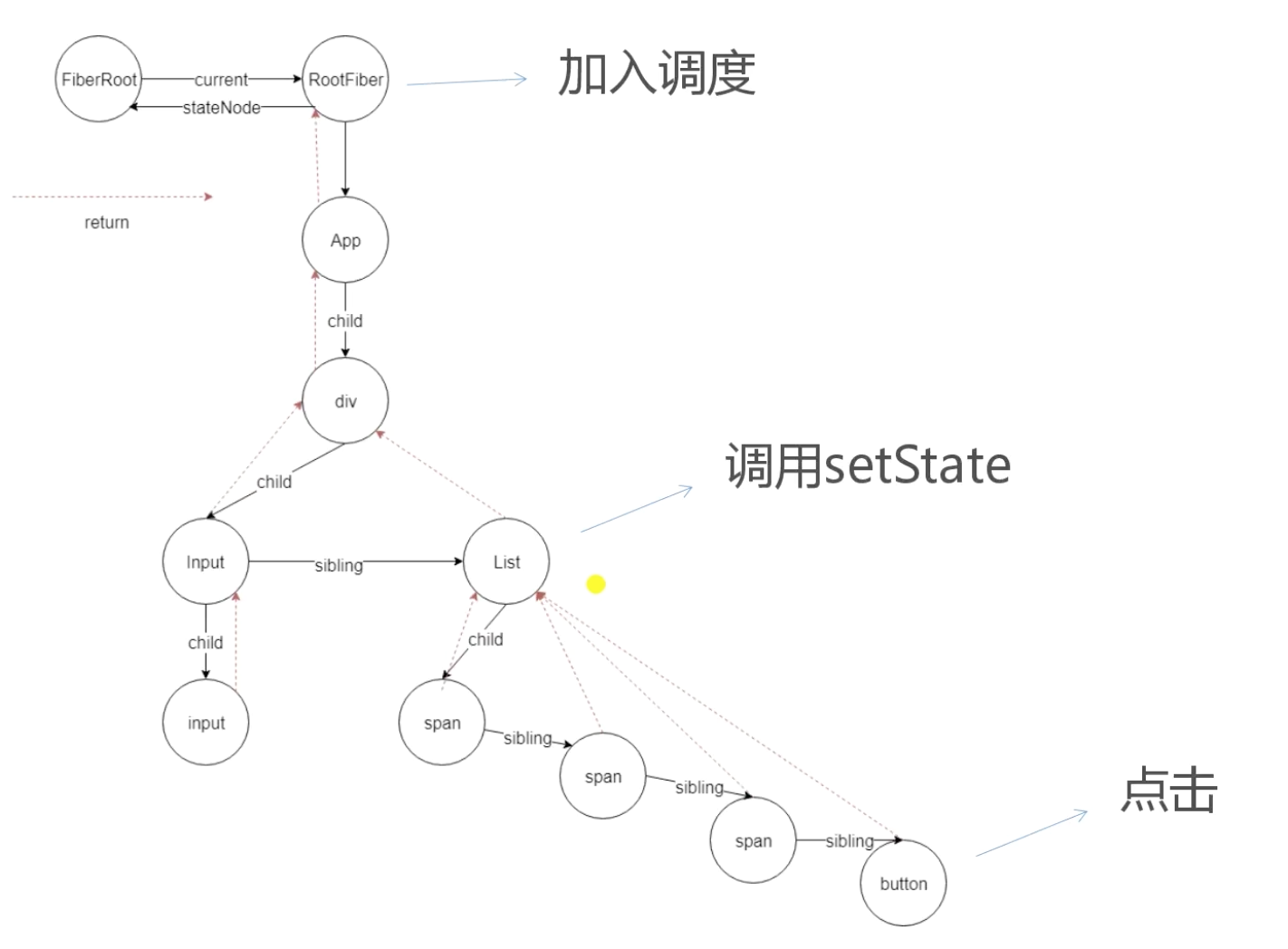
* 调度过程中的各种变量一览
* 构建任务调度的概念

调度什么东西，以及为了达到什么样的目的

1. scheduleWork

* 找到对应的FiberRoot节点：

调用ReactDOM.render时候，传给scheduleWork的是RootFiber节点，在使用setState和forceUpdate时候，传过去的都是某一个组件对应的Fiber节点，这时候我们要找对应的RootFiber节点。



每一次进入调度队列的都是RootFiber节点，不会是其他的，更新的开始，也是RootFiber。

* 如果符合条件重置stack：

stack里面存储一些公共变量，

* 如果符合条件就请求工作调度：

1. requestWork

* 把root节点加入到调度队列当中；
* 判断是否是批量更新；
* 根据expirationTime类型判断调度类型

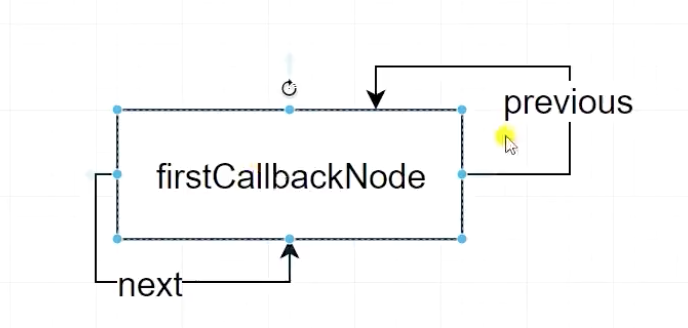
1. batchedUpadtes

在一个方法调用里面，所有的setState都一次跟新到state里面的。

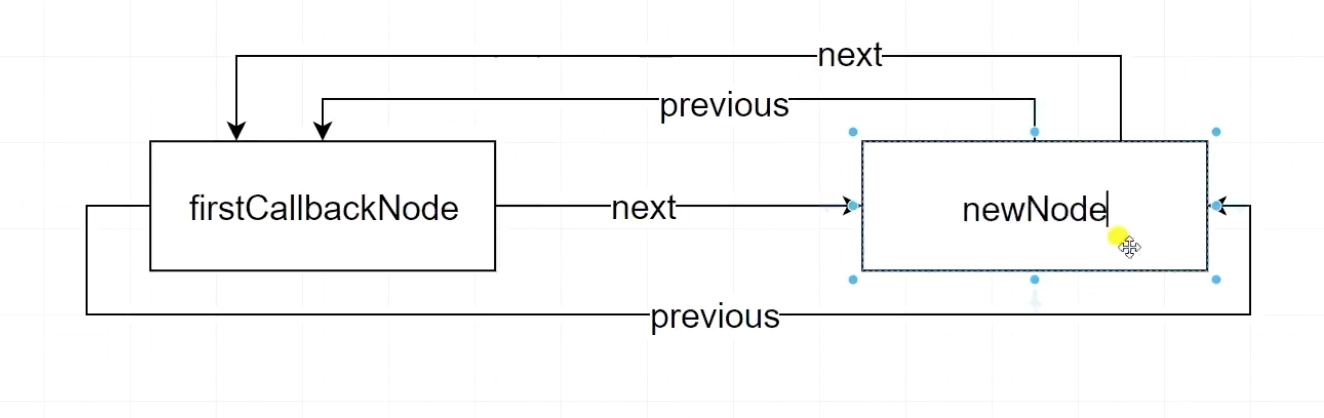
setState是同步还是异步：

setState本身的方法调用是同步的，调用了setState并不标志着state立马就更新了，这个更新是要根据当前上下文的执行环境来判断的，如果处理批量更新的情况，state不是立马就更新的，而不处理批量更新的情况，有可能是立马更新的，如果处于异步渲染情况，那也不是立马更新的。

1. reactScheduler:异步调度



node中只有一个node时候；



多个node的时候

* 维护时间片
* 模拟requestIdleCallback:

传入一个回调，等浏览器有空闲的时候再执行回调的内容

* 调度列表，进行超时判断

1. performWork

同步任务执行的过程，怎么安排root节点链条的更新，又怎么去更新一个fiber树，他的每个节点如何遍历到和更新操作的。

* 是否有deadline的区分:

执行performWork过程中，是否有deadline（通过react的schedule的时间片更新，产生的dealineObject对象，可以用来判断一帧的渲染过程中，留给react渲染树渲染的时间还有没有）的区分。

* 循环渲染Root的条件：

一个应用过程中，可能会有多个root节点，每一个root节点上，可能会有多个优先级不同的任务产生，我们要循环便利各个不同的root节点，以及各个不同优先级的任务，然后按照优先级，一个一个更新他，这个循环如何建立，如何判断这个循环是否成立的条件，这也是一个核心知识。

* 超过时间片的处理：

deadline到了之后，我们这一帧的渲染时间已经到了，我们需要把js的执行权交给浏览器，这个时间我们又应该怎么做。

1. renderRoot

* 调用workLoop进行循环单元更新

遍历整个fiber Tree,把每一个组件或者DOM节点对应的fiber节点拿出来，单一的进行更新，这是一个循环的操作，我要把整个fiber Tree都遍历一遍。

* 捕获错误并进行处理

在进行每一个单元更新的时候，遍历逻辑的时侯，有可能出现一些错误，有些是可预期的，有一些是不可预期的，需要对他们做一些处理，要告诉react，这里出现了一些错误，以及如何对他做一些处理。

* 走完流程之后进行善后

流程走完之后，会出现一些错误的情况，有任务被挂起的时候，按照特定的逻辑进行处理。

1. 最后补充

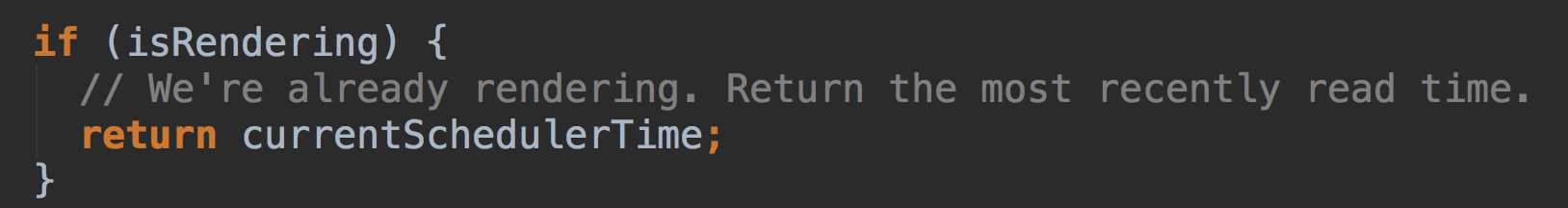
* currentTime

计算expirationTime时候用到，表示指定的当前时间，在react的整体设计中，是有一些用处和特殊的设定的，

* 在一次渲染中产生的更新需要使用相同的时间（在这个阶段内，currentTime是相同的）；

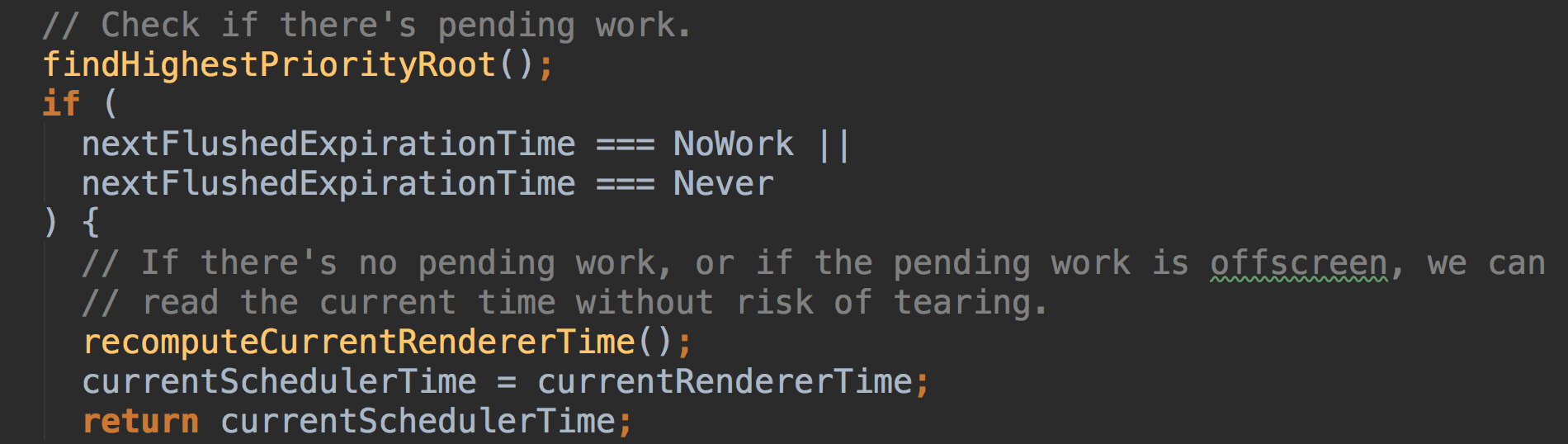
计算requestCurrentTime时候:

一次performWorkOnRoot里面的currentRendererTime是一样的；只有在performWorkOnRoot时候isRendering为true



* 在一次批处理的更新应该得到相同的时间；

在计算requestCurrentTime时候



在多次调用setState时候，第一个setState执行之后，nextFlushedExpirationTime ！== NoWork，进入的条件就不会满足，就不会重新计算currentTime，然后就会返回当前保存的时间，保障在这一个batchUpdate里面用的currentTime是一样的。

* 挂起任务用于记录的时候应该相同。
* expirationTime

在计算expirationTime之前，我们的计算时间是预先处理过的，在误差在10毫秒里面的时间会被磨平。

# 各类组件的Update

1. 入口和优化

对Fiber树每一个节点进行对应的更新。这个入口方法会有一个帮助我们优化整个树更新过程的方法。每一个子节点的更新不会影响兄弟节点的更新，所以需要进行优化的。

* 判断组件的更新是否可以优化
* 根据节点类型分发处理
* 根据expirationTime等信息判断是否可以跳过

调和子节点的过程：在reactDOM.render的过程，只有一个Fiber对象，这个Fiber对象会保存他的props，里面有很多的children，里面有一个children，就是对应的APP对应的reactElement，这个reactElement不是Fiber对象，我们需要创建一个Fiber对象，所以reactDOM.render的过程中，需要一步一步往下过去的，对应于他的children，我们需要判断他的类型，需要不同的处理方式

1. reconcilerChildren

更新了一个节点之后，拿到了他的props.children,根据这个children里面的reactElement来创建他的子树的Fiber对象。

React在更新某一个节点的时候，要根据这个节点的类型，去获取他的children，比如说是functionComponent，他要调用这个Component计算出他的return的属性，return的属性可能是一个数组，可能是单个的reactElement，可能是number，string这些类型，我们要根据这些不同的类型，去进行一些特殊的处理，然后得到他的children，并且对应的Fiber对象，这样我们就可以继续往下迭代，迭代到最终的节点，经过这么一个过程，我们就可以Fiber树进行一个完整的遍历过程。

* 根据props.children生产Fiber子树
* 判断Fiber对象是否可以复用

我们第一次渲染的时候，就已经渲染了整个Fiber子树，在有一个更新进来之后，state变化可能会导致有一些子节点产生变化，可能就不能复用之前的节点了，大部分的fiber节点是可以进行复用的，这个时候是根据什么进行判断。

* 列表根据key优化

key有什么用处，有哪些优化的点

1. reconcilerChildren-array

在react更新某一个节点的时候，要根据这个节点的类型，去获取他的child，比如functionComponent，他要调用这个Component，计算出他return的属性，return的属性可能是一个数组，可能是单个的reactElement，可能是Number，string，我们要根据这些不同的类型，去进行一些不同的处理，然后得到children，并且对应的Fiber对象，这样我们就可以继续往下迭代，然后迭代到最终的节点，经过这个过程我们就可以把一个Fiber树一侧的子树完整的完整的遍历和构建的过程。

* key的作用
* 对比数字children是否可复用
* generator和Array的区别

1. classComponent

组件更新，承担了唯一一个可以更新整个应用的API，在应用中，只有更新了state，才会重新渲染。

1. HostRoot

是一个特殊的节点，在react中，只有一个hostRoot，对应的Fiber对象就是RootFiber对象

1. HostComponent

HostComponent就是DOM原生的节点，就是在JSX编写的时候，用小写写的内容

1. HostText  
   单纯的文本节点
2. ForwardRef

ForwardRef里面的reactElement类型里面的Type是一个对象，该对象里面的$$typeof是FORWARD\_\_REF,他有一个render,就是传进去的functionComponent

1. Mode

Mode组件都是react提供给我们的原生组件

一个是：ConCurrentmode;

一个是：StrictMode

1. Memo

创建一个具有purComponent特性的functionComponent

# 完成节点任务

1. completeUnitOfWork的整体流程和意义

* 根据是否中断调用不同的处理方法

中断：渲染一个组件的时候，出现错误，会给他执行一定的标记，会对不同标记的错误执行不同的方法

* 判断是否有兄弟节点执行不同的操作
* 完成节点之后赋值effect链

把所有的更新节点挂载到真正的DOM上面，这个阶段叫commit阶段，他要去执行每个节点不同的跟DOM有关的操作，是在我们前面beginWork和completeWork对每一个节点更新过程标记的是有哪些setEffect最终是要被commit的过程，所以在completeUnitOfWork里面，会把所有的effect节点进行串联，让commit阶段可以非常的方便根据这个链执行每个节点的操作。

1. 重设childExpirationTime

childExpirationTime用来某一个节点他的子树当中目前优先级最高的那个更新。因为我们整个应用调度过程中使用的都是root节点，我们在scheduleWork时候，即便我们创建更新的节点是某一个组件，但是最终我们要先找到root节点，我们再把他放到调度队列当中。当子树多的时候，我们可以通过childExpirationTime，在root节点上找到优先级最高的更新的节点。

当把优先级最高的节点更新完成之后，需要对他的父节点已经至上的所有节点的childExpirationTime进行更新

1. completeWork具体做了什么，

completeWork我们在completeUnitOfWork当中，在节点是正常渲染，没有任何出错的情况下，我们会去调用completeWork对这个节点进行一个完成工作，对这个节点完成一些操作。

* Pop各种context相关的内容
* 对于HostComponent执行初始化
* 初始化监听时间

1. 初次渲染中completeWork对于DOM节点的创建和appendAllChild算法

* diffProperties计算需要更新的内容
* 不同的dom property处理方式不同

1. 初次渲染中如何进行DOM节点属性初始化操作
2. 更新DOM时进行的diff判断

virtualDOM是一个什么样的东西：就是通过props进行一个attribute的判断，是否有变化

1. completeWork阶段对于HostText的更新
2. renderedRoot中对于错误的处理

* 给报错节点增加Incomplete副作用

Incomplete副作用对于我们在completeUnitOfWork时候，就拿他来判断，我们是调用completeWork，还是UnwindWork

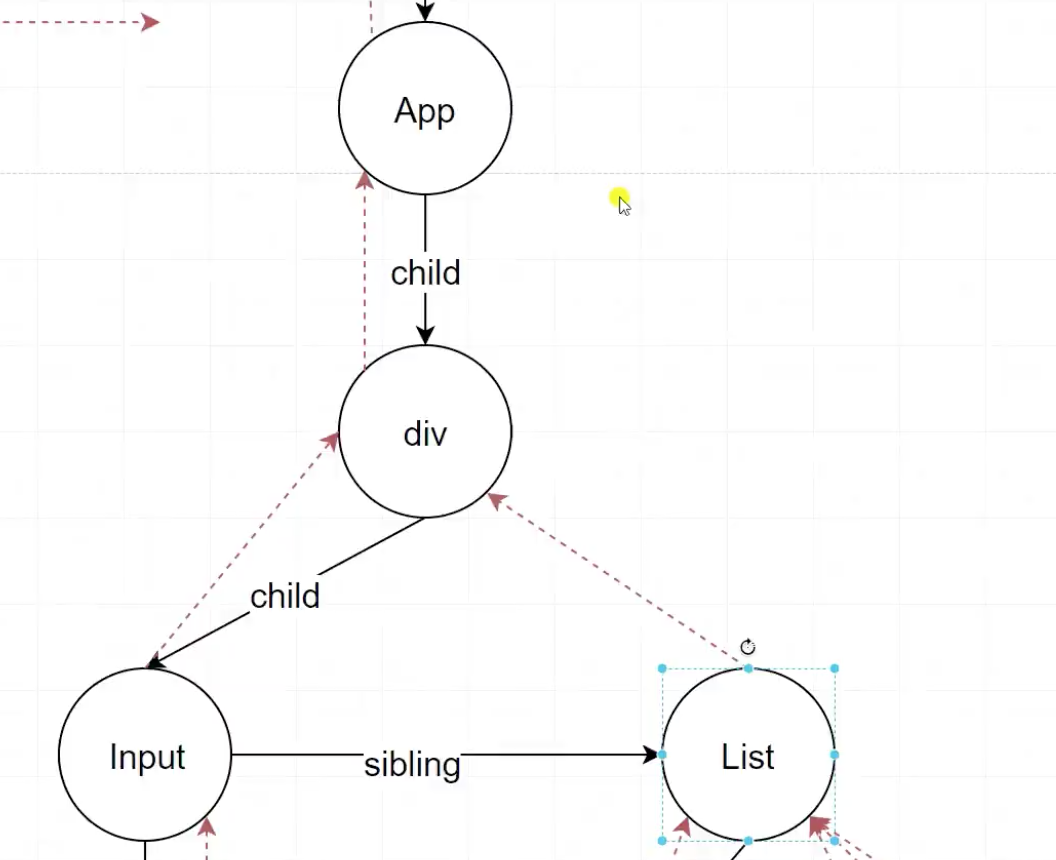
* 给父链上具有error boundary的节点增加副作用

让他们为我们收集错误以及错误处理

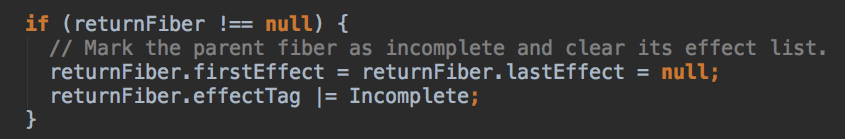
* 创建错误相关的更新

1. unwindWork以及react中的错误处理

* 类似completeWork对不同类型组件进行处理
* 对ShouldCapture组件设置DidCapture副作用

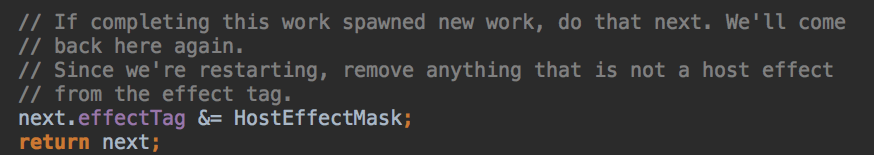


如果List组件渲染时候报错，但是他没有getDerivedStateFromError，componentDidCatch这样的生命周期方法，他是不会增加shouldcapture这个sideEffect的，如果这个组件App组件具有的getDerivedStateFromError，componentDidCatch那么这个时候，App增加的是shouldcapure这个sideEfffect，在我们throwException完之后，我们要执行的第一个completeUnitOfWork是list这个组件，在执行unwindWork时候，return的next是没有内容的，然后会走这条路



然后还会走completeUnitOfWork(这时候是div组件),然后仍然走的是unwindWork,div是HostComponent组件，这个时候不会处理错误，然后又往下走，走到App，这个时候是有ShouldCapture,这个时候，return的是workInProgress，这个时候就能进入这个条件里面

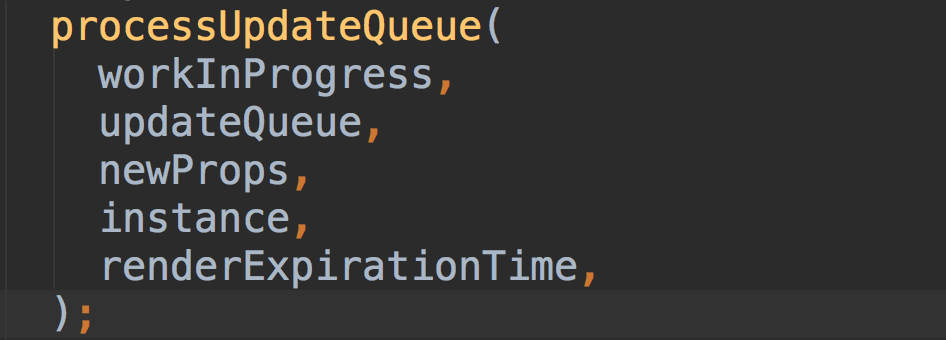




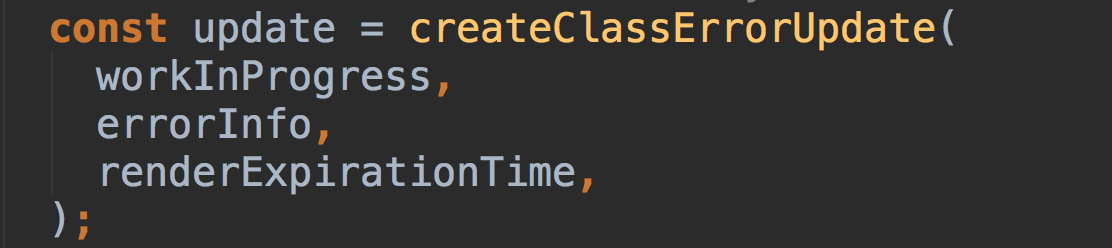
return了next之后，对于completeUnitOfWork，就相当于return了一个Fiber对象，



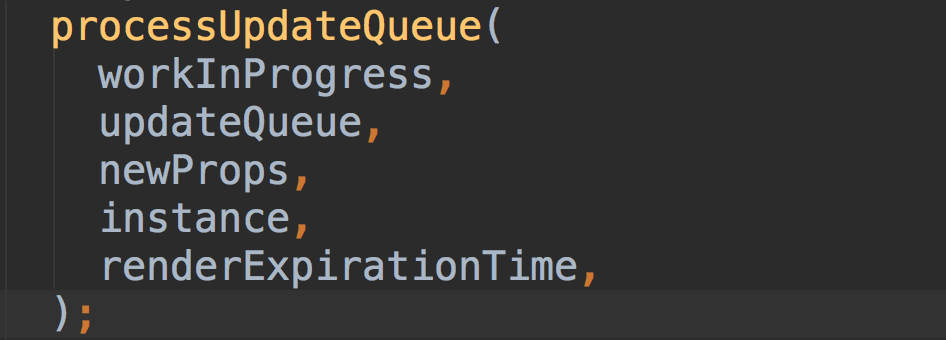
这个时候，nextUnitOfWork返回的app这个组件Fiber对象，do while循环仍然继续，调用workLoop，调用performUnitOfWork,调用beginWork，所以对应app这个组件，就需要重新走一遍更新的流程，这个时候，更新的流程会和之前不一样，走到updateClassComponent时候，比如在创建updateClassInstace时候



因为我们在unwindWork时候，给他创建了一个error的update，



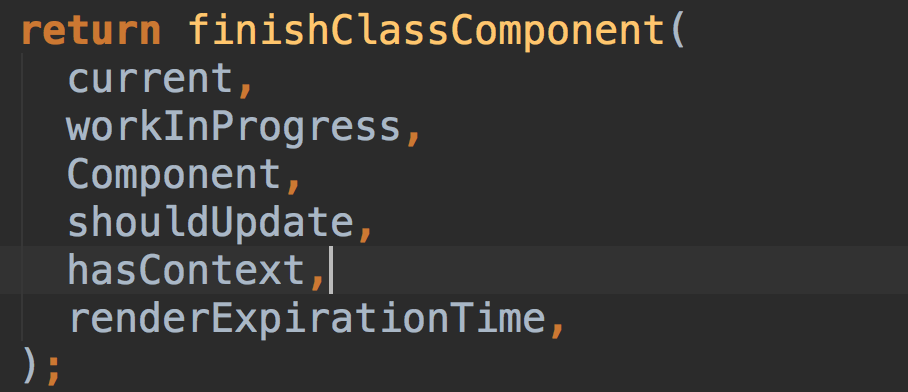
这个update会去调用getDerivedStateFromError,以及他有一个callback会调用componentDidCatch，这个时候我们在调用



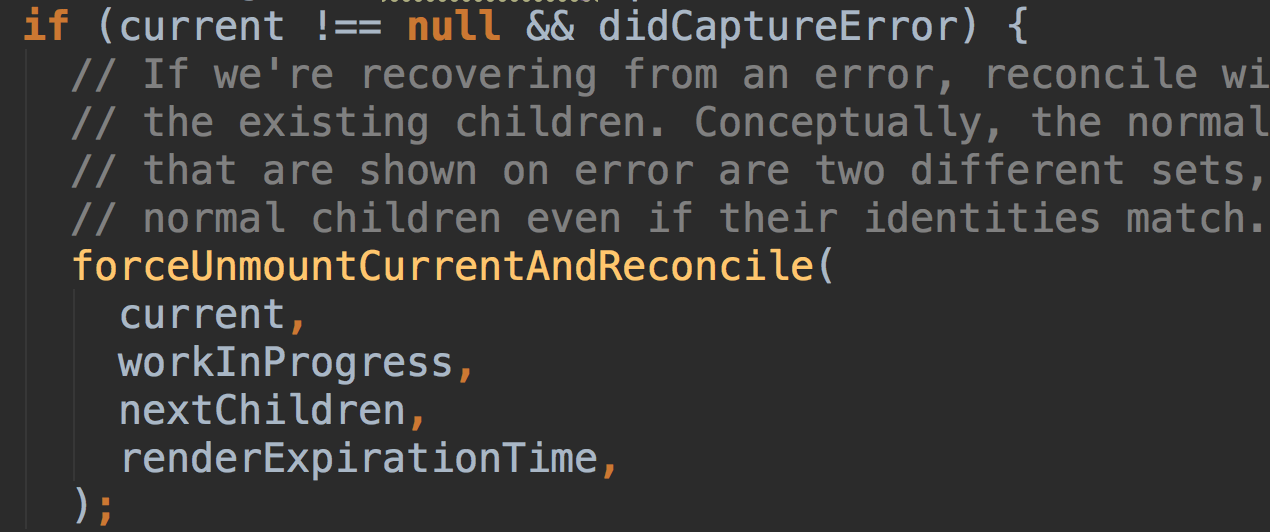
的时候，肯定会调用这个方法，那么他就会计算出有错误信息的时候的state，这个state就会引导classComponent去渲染错误相关的UI。

这个就是组件捕获错误，渲染错误UI的流程，因为渲染的是错误的UI，原先的子树就不会被渲染，或者有可能会被选入出来，但是这就是一个有区别的情况。

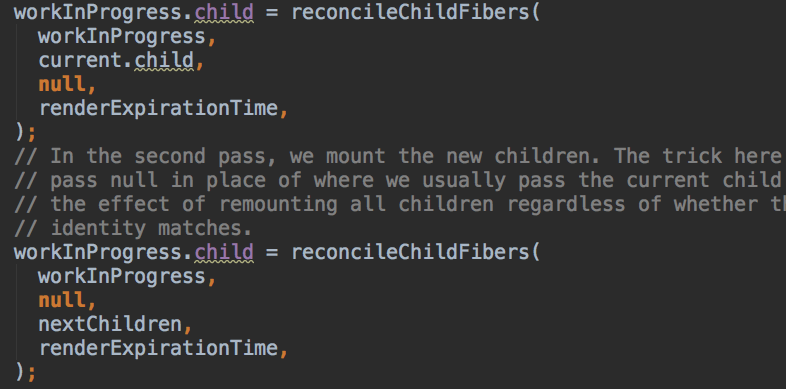
在updateClassComponent的



会进入这个条件



就进入这样一个情况



先调用reconcileChildFibers，但是传入的nextChildren是null，要强制将现在所有子树的节点进行删除掉，渲染出一个没有子树的classComponent，然后再渲染一遍，这个时候传入的是老的children是null，然后传入一个nextchildren，这个时候的nextChildren是对错误捕获并计算出的nextChildren，这个nextChildren跟之前的nextChildren是不一样的。

# CommitRoot

之前讲解的是React当中的render阶段:

进行一些列的调度，一些列的节点的更新过程，重新计算他的state，props，生成新的DOM树，以及我们的Fiber树，这个过程中，每一个节点的更新过程都是独立的，每一个节点更新完之后，都可以跳出更新的循环，然后根据不同的更新模式，可以被中断，可以分片的进行更新，能够让react把够多的优先权交给浏览器，让他可以及时的更新用户的反馈，以及一些持续性的动画，让我们整体的页面变的更流畅。

Commit阶段：

整个流程是不可以被中断的，但是会把尽量少的任务交给commit阶段，在这个阶段，需要更新的节点尽可能的少。

在renderRoot结束的时候，在root上面设置finishedWork这样一个对象，这个对象是RootFiber对象，Rootfiber对象会包含了一个叫firstEffect到lastEffect这么一个链，这个链就包含了所有在所有的render阶段计算出来的需要需要在commit阶段进行一些操作的节点，这些节点是在commit阶段根据赋值给他们的不同的sideEffect来进行一些不同的操作。

1. commitRoot的总体工作内容

commitRoot这个方法他总体的一个流程，这个方法里面包含很多处理sideEffect的内容。

* 预备工作
* 三个循环：

这三个循环都是对firtstEffect到lastEffect单向链表上面，每一个节点上面去跟新的内容

* 善后工作

1. invokeGuardedCallback开发时的帮助方法
2. commitRoot第一个操作-获取快照

获取状态快照，用于componentDidUpdate

1. commitAllHostEffects总体做了哪些事情
2. commitPlacement插入新的子节点的操作

把一个新的DOM节点挂载到DOM树上的一个过程

1. commitWork更新节点属性的过程
2. commitWork删除节点的操作过程

* 遍历子树：

对于react组件树来说，一个DOM节点下是可以存放classComponent这样的节点的，我要删除这个节点的同时，相当于也要删除这个component，这个classComponent如果有生命周期方法（componentWillUnmount），我们要提醒他，要调用这个方法，如何知道有没有，我们就需要遍历这个子树。如果存在portal这样组件，需要把他从container里面去删除。

* 卸载ref

这个DOM上面如果挂载了ref这个属性，我们在render这个DOM节点他的owner上面，他的component上面，就有某个ref属性是指向这个DOM节点的，如果我们已经把这个DOM节点删掉了，这个ref如果还指向这个DOM节点，那么肯定是不对的。

* 如果有组件调用componentWillUnmount方法

1. commitLifecycles调用生命周期方法

# 功能详解：基础

创建一个更新，到进入调度，更新每一个组件节点，形成一个完整的fiber树，把这一个Fiber树跟前一个Fiber的对比结果，到commit阶段，把这些更新的内容，放到DOM上面，形成一个UI的更新，这个内容涵盖了我们写代码的过程操作，到react内部真正的去更新完成，最终把我们更新的state之后，应该现实的UI给显示出来的一个完整的过程。在react中，我们只能通过setState设定一个状态，然后通过这个状态，去映射一个样式，来达到这样的目的。

1. context-stack

react中有唯一的一个stack，里面保存了各种个样的东西。

Stack栈是一个先入后出，对应于react更新过程中，我们的beginWork是从头到尾沿着子树的一侧下来的过程，在completeUnitWork时候，我们是从尾到头完成节点更新到操作。

保障更新开始到更新结束的时候，栈是空的。

* 更新节点时，相关信息入栈
* 完成节点时相关信息出栈
* 用不同的cursor记录不同的信息

1. 遗留context-api的实现过程

* 会影响整个子树更新过程
* 嵌套的context提供者需要进行合并
* 对性能影响很大

1. 遗留conetx-api的实现过程
2. 新context的实现

* 组件化的使用方式

像jsx一样使用

* context的提供方和订阅方都是独立的
* 没有什么附带的性能影响

我们在更新Proviter的时候，给context.\_currentValue设置了一个值，设置的值是新的context提供的value，以此让我们的consumer获取到这个值，对于provider更新了之后，如何去通知他的consumer依赖的组件，我们需要手动的去遍历他的所有子节点，然后去指定他的expirationTime，然后告诉他的更新流程里面，依赖的context需要进行更新。

1. ref的实现过程

* 创建Fiber的时候处理ref

什么时候创建fiber对象：我们去更新某个节点，然后要调和他的子节点，这个时候，我们会对每个子节点创建fiber对象，在创建fiber对象的时候，就会去处理ref

* 在commit开始先detach

1. hydrate-是否需要hydrate的判断

在第一次渲染的时候，如果DOM树上面本身有一个DOM结构的存在，是否可以利用这一部分的存在的DOM，去避免掉我们第一次渲染的时候，创建很多DOM节点的过程，可以大大提升第一次渲染的过程

1. hydrate-再新开始判断节点是否可以hydrate
2. hydrate-再completeWork中复用可hydrate的节点
3. event事件系统初始化-注入平台时间插件

在初始化reactDOM源码的时候，我们会为react事件系统去注入DOM相关的时间，react事件系统会有一个独立的模块，这个模块是一个公用性质的模块（reactDOM和reactNative都可以用）

* 确定插件注入顺序
* 诸如插件模块
* 计算registrationNameModules等属性

1. event事件监听的过程
2. event-事件触发的过程
3. event-事件对象的生产过程
4. event查漏补缺

* 关于batchedUpdates

我们通过事件去触发的大部分setState都处于batchedUpdates里面，也就是这一部分的setState跟新都会一起被提交，最终在一起被回调。

* Controlled inputes如何回调
* 关于事件绑定的dispatchEvent

# suspense and priority

react在更新的过程中，任务是可以被挂起的，什么叫suspense，优先级可以分为几类。

1. 优先级和任务挂起的含义

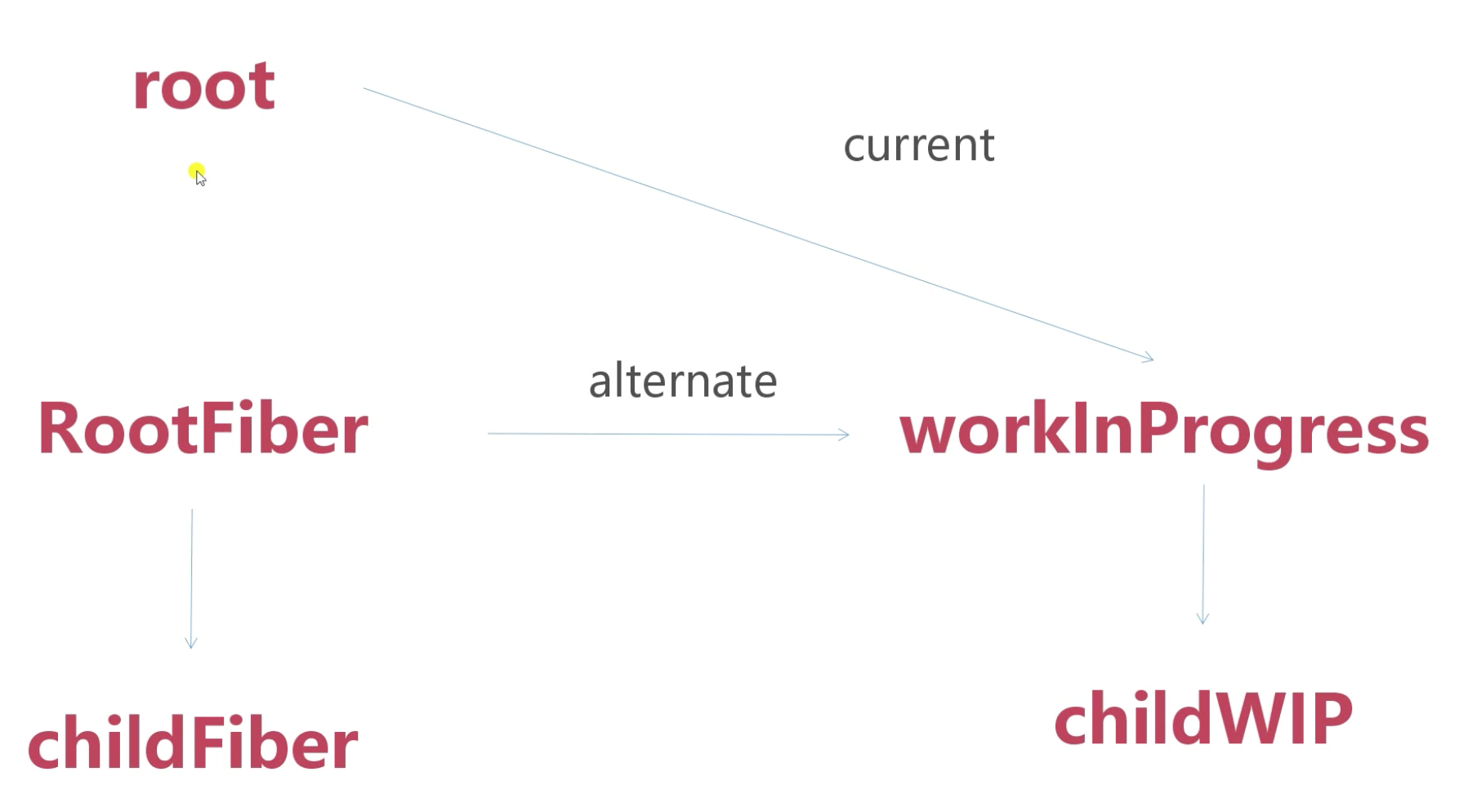
* 字面意思是挂起
* 某次更新的任务暂时不提交

react更新分为两个阶段，一个是render阶段，主要是包含我们performUnitOfWork和completeUnitOfWork，我们对拿到的reactElement一层一层向下渲染，这个过程叫做beginWork，这个过程中我们从一个root节点开始渲染，渲染到某一层的最终子节点之后，再由这个子节点向上返回，并且渲染他的兄弟节点，最终回到root这个过程,把一个reactElement形成的树，最终渲染成一个fiber树，这个过程中，我们会更具传入的props以及节点的每个不同的类型来给他创建不同的实例，以及进行一些default的内容，这个过程叫做render过程，这个过程为什么叫render过程，因为他完全不会影响目前的整个DOM树，他不会更新DOM节点的任何特性，我们在更新的过程会形成一个Effect链，这些effect链代表着我们在commit阶段，是把哪些DOM节点进行更新，这个阶段完成之后，我们进入commit阶段，把我们render阶段能够发现的所有需要更新的节点提交到DOM上面，来完成UI的更新。

在suspend当中，我们完成了所有的render任务，我们暂时把所有的任务停在render阶段，不提交，也就是把最终的需要修改DOM的任务给他停掉，这个更新过程就叫被suspend了。

* 这个更新可能在下一次更新中再次被执行

任务更新完之后，重新改变root的current指向



* 三种suspend的方式

把提交放到底优先级的任务上

* 直接发起一个新的同步更新

强制发起一个同步任务，重新render看错误是否能解决，不能解决按照错误的方式进行提交

* 设置timeout然后提交

会设置timeout情况只有throw的promise的情况，就是通过suspense这个功能，throw promise，等到promise解决了之后，再去渲染新的内容一个情况

* 类型
* pendingTime
* suspendedTime
* pingedTime

1. 两个expirationTime的不同作用

* expirationTime作用在渲染之前
* nextExpirationTimeToWorkOn作用在渲染时

1. suspense组件同步模式下的更新

* 先直接渲染子节点为null
* commit的时候设置state
* 再发起一次同步更新渲染fallback

1. suspense组件同步渲染模式补充
2. suspense组件一步模式下的更新

* 设置shouldCapture
* unwindWork设置state
* 渲染fallback

1. retrySuspendedWork所做的事情

# 功能详解：hooks

赋予function组件具有类组件能力。

能够更好的提取业务逻辑出来，能够更多的复用到组件上面。

useState:state、setState

useEffect:使用的生命周期方法

useContext:获取context

useReducer：useState的本质是useReducer

useLayoutEffect:其实和useEffect其实是差不多的

useCallback:帮助我们去优化，我们在渲染过程中出现的重复渲染

useMemo: 帮助我们去优化，我们在渲染过程中出现的重复渲染

useRef:提供具有ref的功能

useImperativeHandle：