

Fiber

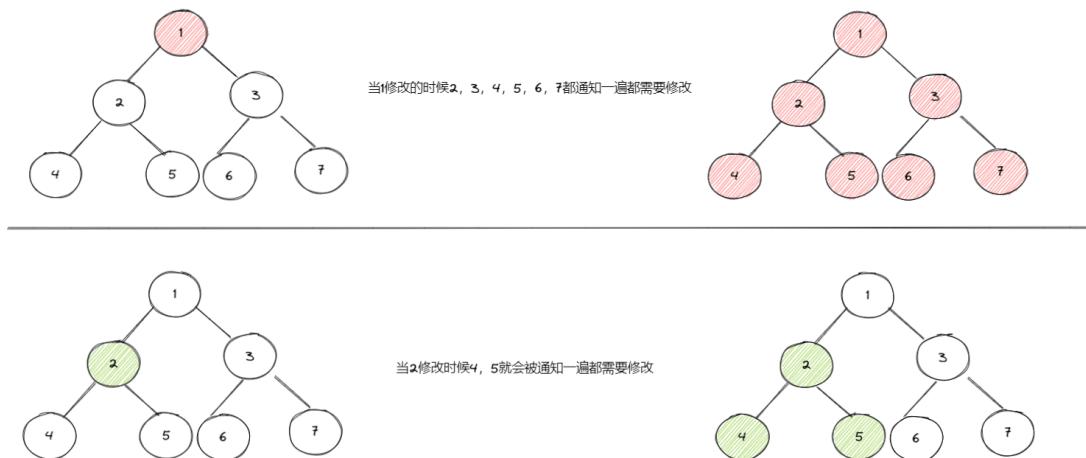
初始

react16之前 它是渲染方式非常低(自上向下), 整体渲染, 渲染是一次性的, 权重一样, 这样在数据少的时候, 它没有感觉, 但是如果你的数据一多, 在渲染时就会有卡顿。react16之前渲染效果: <https://claudiopro.github.io/react-fiber-vs-stack-demo/stack.html>

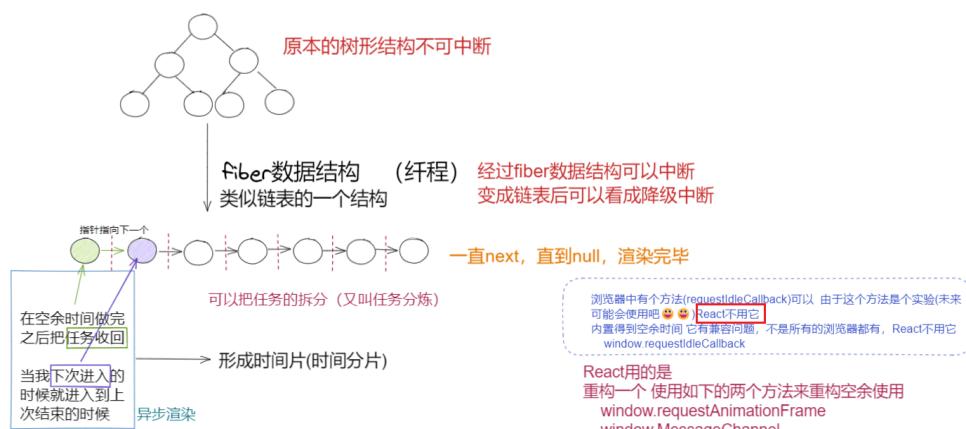
react16之后, 引入一个fiber数据结构, 对于渲染进行任务拆分, 让渲染变成多任务方式, 利用浏览器的空余时间来进行渲染, 但是它有了增量渲染和优先级。(时间分片, 任务分炼) react16之后引入fiber后渲染效果: <https://claudiopro.github.io/react-fiber-vs-stack-demo/fiber.html>

通过上面两个链接展示, 可以发现react在没有引入fiber渲染方式时, 它的渲染会有卡顿, 不流畅, 引入后渲染更加流畅, 事件响应更加及时

渲染方式中vue比react更快, vue更加高效, vue做了代理(对所有数据经行渲染劫持, 就可以做到精准渲染), 对于React没有做数据代理, 两者之间React对于数据更加高效, 对于渲染没有vue高效, 在React中一个数据的更改会影响到它下面的所有数据都要改掉, React的渲染自上向下如下图所示(React16之前)



但是在React16之后引入了fiber数据结构, 利用浏览器的空余时间来进行渲染(增量渲染和优先级)方式如下



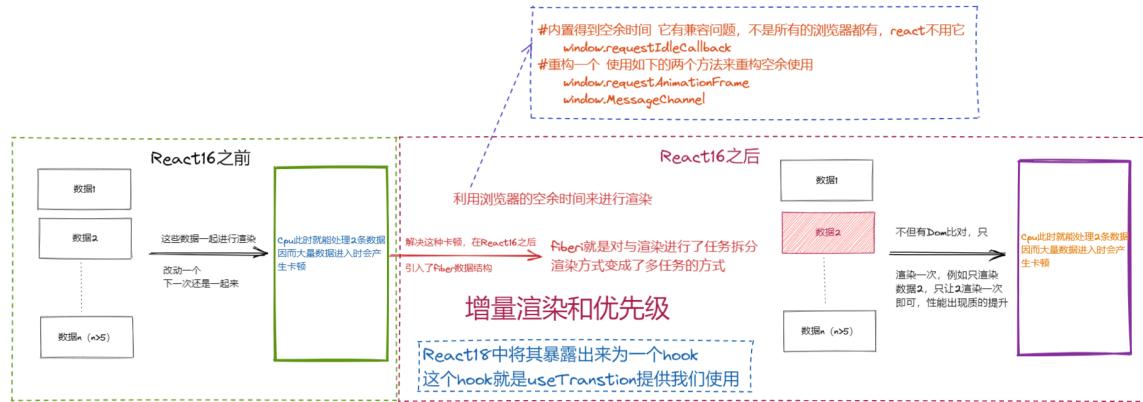
#内置得到空余时间 它有兼容问题，不是所有的浏览器都有，react不用它

window.requestIdleCallback

#重构一个 使用如下的两个方法来重构空余使用

window.requestAnimationFrame

window.MessageChannel



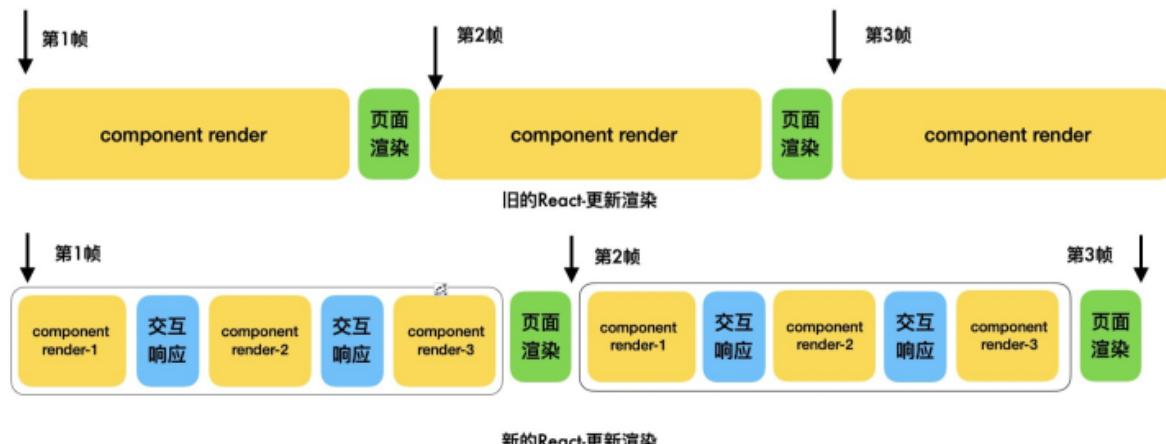
帧的概念

- 目前大多数设备的屏幕的刷新率为60次/秒，即60帧每秒，人眼舒适放松时可视帧数是每秒24帧，帧数(fps)越高，所显示的动作就会越流畅，小于这个值的时候，用户就会感觉到卡顿，所以对于现在主流屏幕设备来说，每个帧的预算时间就是 $1/60$ 约等于16.66毫秒
- 每个帧的开头包括样式计算、布局和绘制
- JavaScript执行Javascript引擎和页面渲染在同一个线程中，GUI渲染和Javascript执行两者之间是互斥的

[浏览器的线程](https://blog.csdn.net/Mabius/article/details/122898389)(<https://blog.csdn.net/Mabius/article/details/122898389>)

1. GUI渲染线程
2. 定时器线程
3. 事件监听线程
4. js引擎线程
5. http网络线程

4. 如果某个任务执行时间过长，浏览器就会推迟渲染。



要实现Fiber架构，必须要解决两个问题：

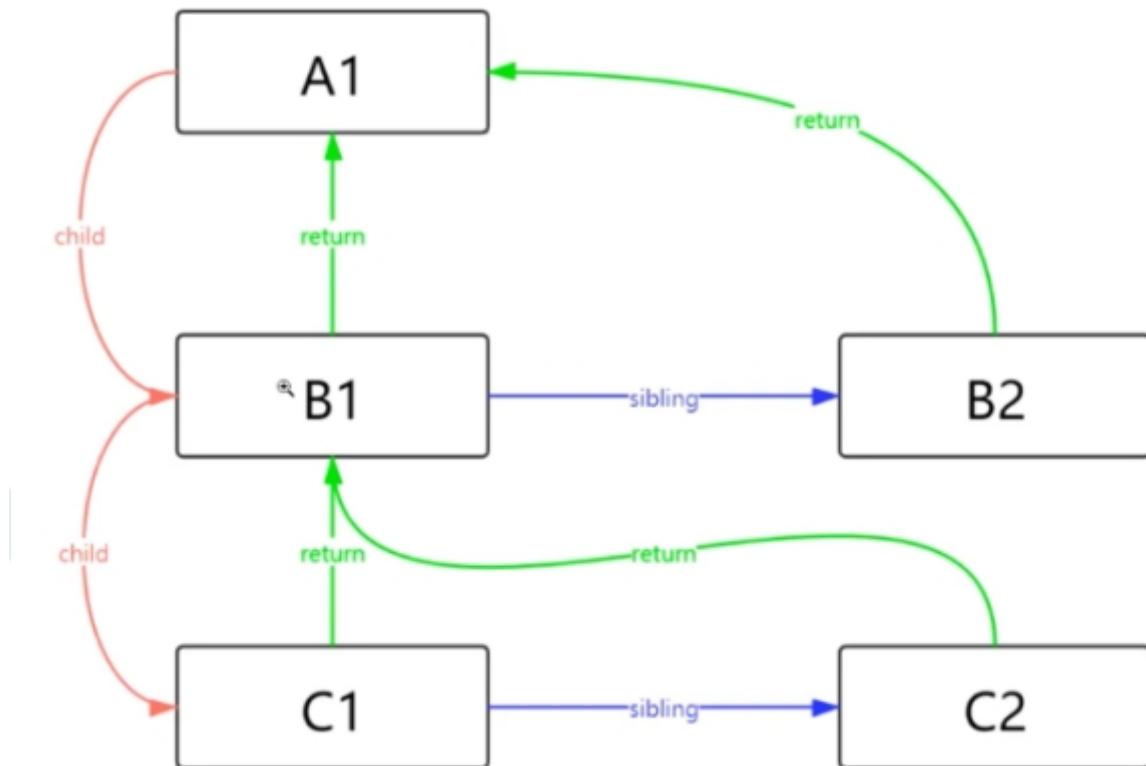
1. 保证任务在浏览器空闲的时候执行；`requestidlecallback`(React中不使用，但是是一个思想)，用的是`requestAnimationFrame+MessageChannel`

2. 将任务进行碎片化; `链表(fiber)`

Fiber 结构：链表结构

目前的虚拟DOM是树结构，当任务被打断后，树结构无法恢复之前的任务继续执行，所以需要一种新的数据结构，即链表，链表可以包含多个指针，可以轻易找到下一个节点，继而恢复任务的执行。

Fiber采用的链表中包含三个指针，`return`指向其父Fiber节点，`child`指向其子Fiber节点，`sibling`指向其兄弟Fiber节点。一个Fiber节点对应一个任务节点。



Fiber执行阶段

每次渲染有两个阶段：`Reconciliation`(协调render阶段) 和 `Commit`(提交阶段)

协调阶段

协调阶段：可以认为是 `diff` 阶段，这个阶段可以被终止，这个阶段会找出所有节点变更，例如节点新增、删除、属性变更等等，这些变更React称之为副作用(`EffectList`)。虚拟DOM转成 `Fiber` 链表的阶段，协调阶段是可以被打断的阶段。

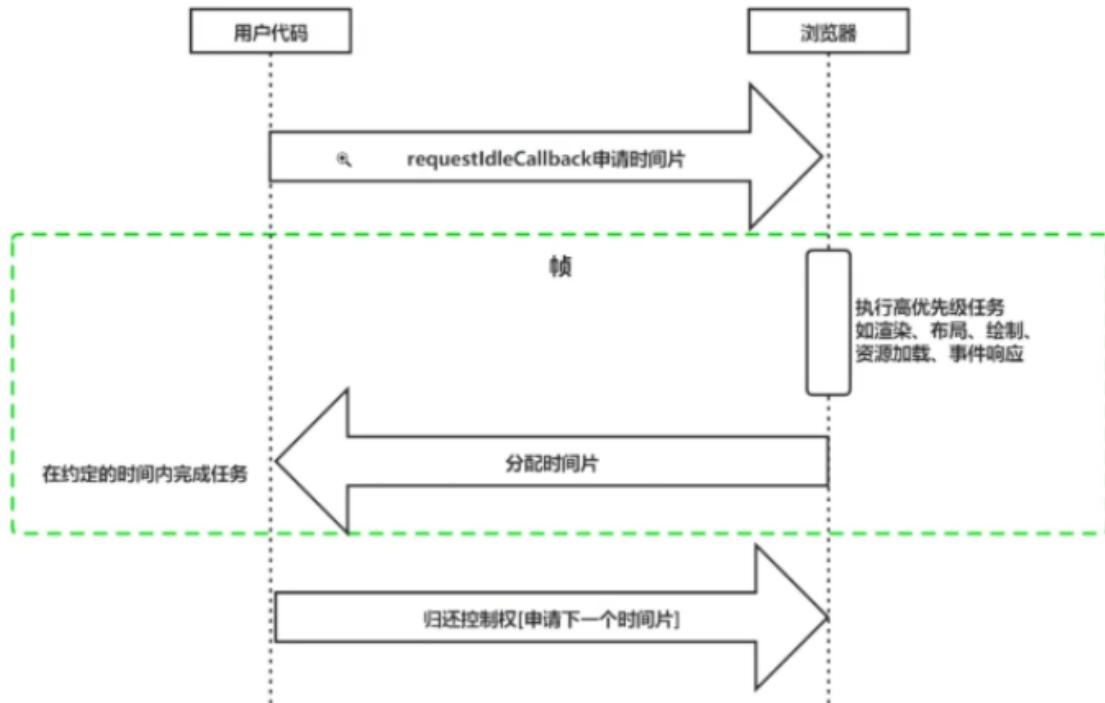
提交阶段

提交阶段：将上一阶段计算出来的需要处理的 副作用(`effects`) 一次性执行了。这个阶段必须同步执行，不能被打断。`把Fiber链表转成Effectlist(副作用链表)----把Fiber链表转成真实DOM渲染到页面的阶段`，这个阶段是不可以被打断的。

requestIdleCallback

[requestIdleCallback - Web API 接口参考 | MDN \(mozilla.org\)](#)

`requestIdleCallback(callback)` 是实验性 API，可以传入一个回调函数，回调函数能够收到一个 `deadline` 对象，通过该对象的 `timeRemaining()` 方法可以获取到当前浏览器的空闲时间，如果有空闲时间，那么就执行一小段任务，如果时间不足了，则继续 `requestIdleCallback`，等到浏览器又有空闲时间的时候再接着执行。



等待时长

```
function sleep(delay = 0) {
    let now = Date.now()
    while (now + delay > Date.now()) { }
}
```

工作任务队列 如下的任务少，没有复杂运算，所以判断再一次空闲时间内就完成

```
let works = [
    () => {
        console.log('任务1 -- 开始');
        sleep(20)
        console.log('任务1 -- 结束');
    },
    () => {
        console.log('任务2 -- 开始');
        sleep(20)
        console.log('任务2 -- 结束');
    },
    () => {
        console.log('任务3 -- 开始');
        sleep(20)
        console.log('任务3 -- 结束');
    }
]
```

工作循环

```
# deadLine它是requestIdleCallback提供的得到空闲时间的形参对象
# deadLine.timeRemaining() 得到空闲时长
function workLoop(deadLine) {
    console.log('剩余时长', deadLine.timeRemaining())
}
```

```

# 只有当我有空余时间时，才执行任务
while (works.length > 0 && deadline.timeRemaining() > 0) {
    # 执行单个任务 执行链表中的一个fiber对象
    performUnitOfWork()
}

# 如果队列中还有任务，则继续要空余时间
if (works.length > 0) {
    window.requestIdleCallback(workLoop)
}
}

# 执行任务，执行时，一定要修改原数组中的数据
function performUnitOfWork() {
    // const work = works.shift()
    const work = works.pop()
    work()
}

# 空余时间
window.requestIdleCallback(workLoop)

```

requestAnimationFrame (动画)

[window.requestAnimationFrame - Web API 接口参考 | MDN \(mozilla.org\)](#)

`window.requestAnimationFrame()` 告诉浏览器——你希望执行一个动画，并且要求浏览器在下次重绘之前调用指定的回调函数更新动画。该方法需要传入一个回调函数作为参数，该回调函数会在浏览器下一次重绘之前执行

注意：若你想在浏览器下次重绘之前继续更新下一帧动画，那么回调函数自身必须再次调用

`window.requestAnimationFrame()`

当你准备更新动画时你应该调用此方法。这将使浏览器在下一次重绘之前调用你传入给该方法的动画函数(即你的回调函数)。回调函数执行次数通常是每秒 60 次，但在大多数遵循 W3C 建议的浏览器中，回调函数执行次数通常与浏览器屏幕刷新次数相匹配。为了提高性能和电池寿命，因此在大多数浏览器里，当 `requestAnimationFrame()` 运行在后台标签页或者隐藏的`__`里时，

`requestAnimationFrame()` 会被暂停调用以提升性能和电池寿命。

回调函数会被传入 `DOMHighResTimeStamp` 参数，`DOMHighResTimeStamp` 指示当前被 `requestAnimationFrame()` 排序的回调函数被触发的时间。在同一个帧中的多个回调函数，它们每一个都会接受到一个相同的时间戳，即使在计算上一个回调函数的工作负载期间已经消耗了一些时间。该时间戳是一个十进制数，单位毫秒，最小精度为 1ms(1000μs)。

```

<!DOCTYPE html>
<html lang="en">

<head>
    <meta charset="UTF-8">
    <meta http-equiv="X-UA-Compatible" content="IE=edge">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Document</title>
    <style>
        #animate {
            width: 100px;
            height: 100px;

```

```

        background: red;
    }

```

```

</style>
</head>

<body>
    <div id="animate"></div>
    <script>
        const element = document.getElementById('animate');
        let start;

        function step(timestamp) {
            if (start === undefined)
                start = timestamp;
            const elapsed = timestamp - start;

            //这里使用`Math.min()`确保元素刚好停在 200px 的位置。
            element.style.transform = 'translateX(' + Math.min(0.1 * elapsed, 200) +
            'px)';

            if (elapsed < 2000) { // 在两秒后停止动画
                window.requestAnimationFrame(step);
            }
        }

        window.requestAnimationFrame(step);
    </script>
</body>

```

MessageChannel(消息通道)

[MessageChannel\(\) - Web API 接口参考 | MDN \(mozilla.org\)](#)

`MessageChannel` 接口的 `MessageChannel()` 构造函数返回一个新的 `MessageChannel` 对象，返回的对象中包含两个 `MessagePort` 对象。

Note: 此特性在 [Web Worker](#) 中可用

语法

```
var channel1 = new MessageChannel();
```

返回值

一个新创建的 `MessageChannel` 对象。

```

<!DOCTYPE html>
<html lang="en">

<head>
    <meta charset="UTF-8">
    <meta http-equiv="X-UA-Compatible" content="IE=edge">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Document</title>

```

```

</head>

<body>
<script>
// 当前页面中完成数据的互通
var channel = new MessageChannel();

// 监听数据
channel.port1.onmessage = e => console.log('port1 -- ', e.data)
channel.port2.onmessage = e => console.log('port2 -- ', e.data)

// port1发送数据给port2
channel.port1.postMessage('来自[port1]的数据')
channel.port2.postMessage('来自[port2]的数据')

</script>
</body>

</html>

```

利用requestAnimationFrame+MessageChannel实现requestIdleCallback

`window.requestIdleCallback` 此方法兼容性较差一些,得到帧空余时间(利用空余时间)

利用如下面的两个方法来完成空余时间的获取

步骤:

1. 得到当前帧它的时长
2. 判断时长是否已过期,没有过期则执行方法(利用空余时间来执行)

`window.requestAnimationFrame` 得到当前帧开始时间 + 每帧的时长 => 当前帧它的时长

`window.MessageChannel` 消息通信,事件编程,声明和处理分开

```

#中间==部分是利用requestAnimationFrame+MessageChannel实现requestIdleCallback
// 等待时长
function sleep(delay = 0) {
  let now = Date.now()
  while (now + delay > Date.now()) { }
}

// 工作任务队列 如下的任务少, 没有复杂运算, 所以判断再一次空闲时间内就完成
let works = [
  () => {
    console.log('任务1 -- 开始');
    sleep(20)
    console.log('任务1 -- 结束');
  },
  () => {
    console.log('任务2 -- 开始');
    sleep(20)
    console.log('任务2 -- 结束');
  },
]

```

```

        () => {
            console.log('任务3 -- 开始');
            sleep(20)
            console.log('任务3 -- 结束');
        }
    ]
=====

# 实现requestIdleCallback
# 1.每帧所用的时长 每秒60帧,1秒在16.66
let activeTimeFrame = 1000 / 60
# 2.结束的时间点 帧开始时间 + 每帧执行时间
let deadFrameTime = 0
# 3.结束执行回调函数 通过事件 再两个方法中,声明一个全局变量,就可以让两法共用此函数
let executecallback
# 4.创建一个消息通信, 帧完成事件和执行的事件不在一起,需在消息通道来通知
const channel = new Messagechannel();
# 5.计算得到当前的剩余时间 performance.now() 获取当前的时间,它的精度更高(计算帧的空余
时间)
const timeRemaining = () => deadFrameTime - performance.now()

# 6.通过消息通信完成数据接受,执行
channel.port2.onmessage = e => {
    # 计算是否有空余时间
    # 得到当前的时间,精确要有小数
    let currentTime = performance.now()
    # 判断它有没有超时,超时返回false,不超时为true
    let didTimeout = deadFrameTime <= currentTime
    # 不超时,且还有空余时间执行方法
    if (didTimeout || timeRemaining() > 0) {# 没有超时
        if (executecallback) {#只有当前执行的函数有值才执行
            executecallback({ didTimeout, timeRemaining })
        }
    }
}

# 重构,当前帧它的空余时间来完成工作
window.requestIdleCallback = function (callback) {
    # 在此帧动画完成后,执行当前的帧中函数(有空余和任务)
    window.requestAnimationFrame((stime) => {
        # 得到当前帧的结束时间点
        deadFrameTime = stime + activeTimeFrame
        executecallback = callback
        # 通过方法执行
        channel.port1.postMessage('')
    })
}
=====

function workLoop(deadLine) {
    console.log('剩余时长', deadLine.timeRemaining())
    // 只有当我有空余时间时,才执行任务
    while (works.length > 0 && deadLine.timeRemaining() > 0) {
        // 执行单个任务 执行链表中的一个fiber对象
        performUnitOfWork()
    }
    // 如果队列中还有任务,则继续要空余时间
    if (works.length > 0) {
        window.requestIdleCallback(workLoop)
    }
}

```

```

}

// 执行任务，执行时，一定要修改原数组中的数据
function performUnitOfWork() {
  // const work = works.shift()
  const work = works.pop()
  work()
}

// 得到帧的空余时间
window.requestIdleCallback(workLoop)

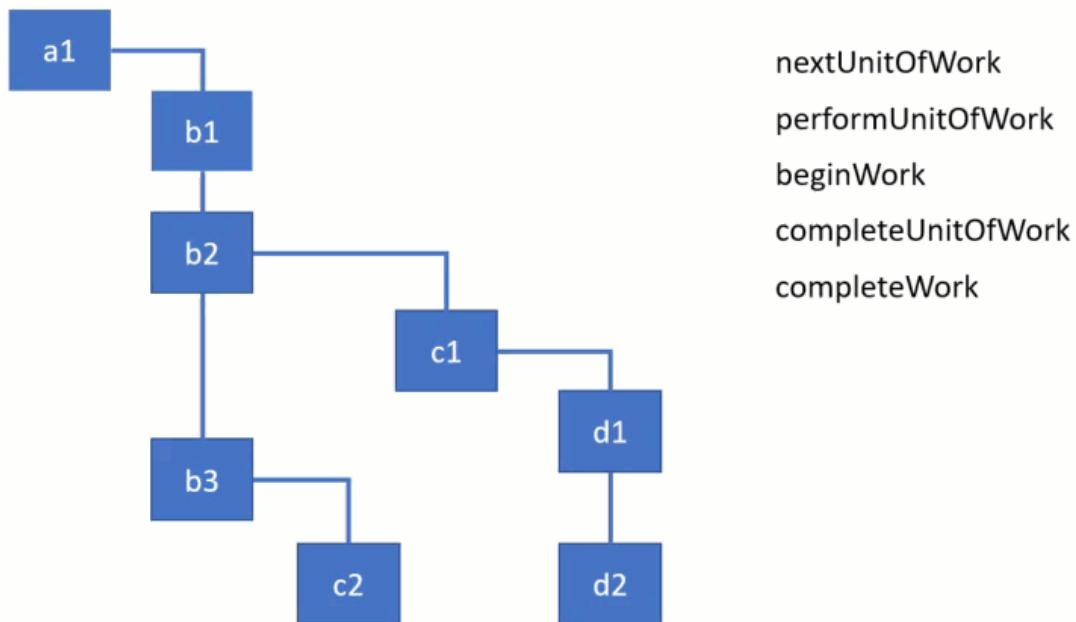
```

实现Fiber结构

深度优先遍历和节点之间是链表关系

首先 Fiber 使用深度优先遍历，来代替之前的递归调用，然后控制深度优先遍历的过程。

1. 从根元素开始，不断遍历，先遍历子元素，看子元素是否有子元素，如果有则继续遍历，如果没有则开始渲染这个元素。
2. 这个元素渲染完之后，检查是否有兄弟元素，如果有则遍历兄弟元素，重复步骤一。
3. 如果没有兄弟元素则返回父元素，处理父元素。
4. 然后一直往上回到根元素，处理根元素。



并且节点之间是一个链表关系，用来模拟函数调用栈。节点记录了它的父节点、子节点和它的最近的兄弟节点。我们可以拿它和函数调用栈对比一下：

	函数调用栈	Fiber
基本单位	函数	fiber node
输入	函数参数	props
本地状态	本地变量	state
输出	函数返回值	react element
下级调用	嵌套函数调用	child node
上级调用	返回地址	return node

如上面的图表所示，fiber node tree 和函数调用栈一样，保存了节点处理的上下文信息，这样我们就可以手动控制节点的渲染过程了。

为 React Element 创建 Fiber Node

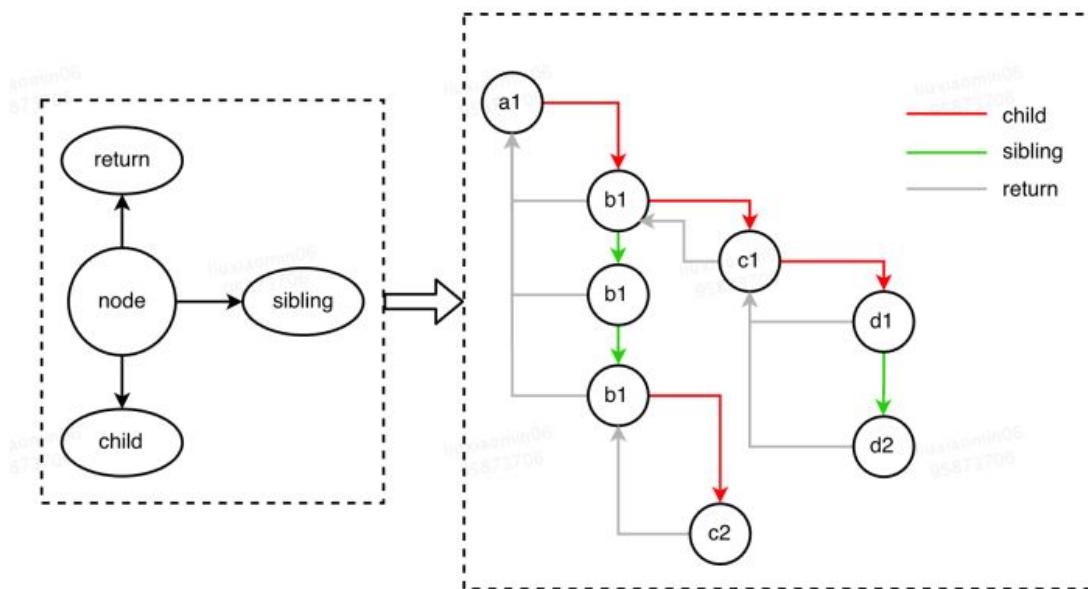
在 reconciliation 期间，会为 render 方法返回的每个 react element 创建一个 fiber node，对应形成了一棵 fiber node 树，在随后的更新中，React 会重用 fiber 节点，并使用来自 react element 的数据来更新自身的属性，如果从 render 方法返回的 react element 有变化，react 会根据 key 来移动或者删除它。这其中相当于改变树节点的数据结构，增加了很多属性：

描述层级关系：

1. `return`，指向父节点。
2. `child`，指向第一个子节点。
3. `sibling`，指向下一个兄弟节点。

通过节点上的 `child`（孩子）、`return`（父）和 `sibling`（兄弟）属性串联着其他节点，形成了一棵 Fiber Tree (类似 virtual DOM tree)

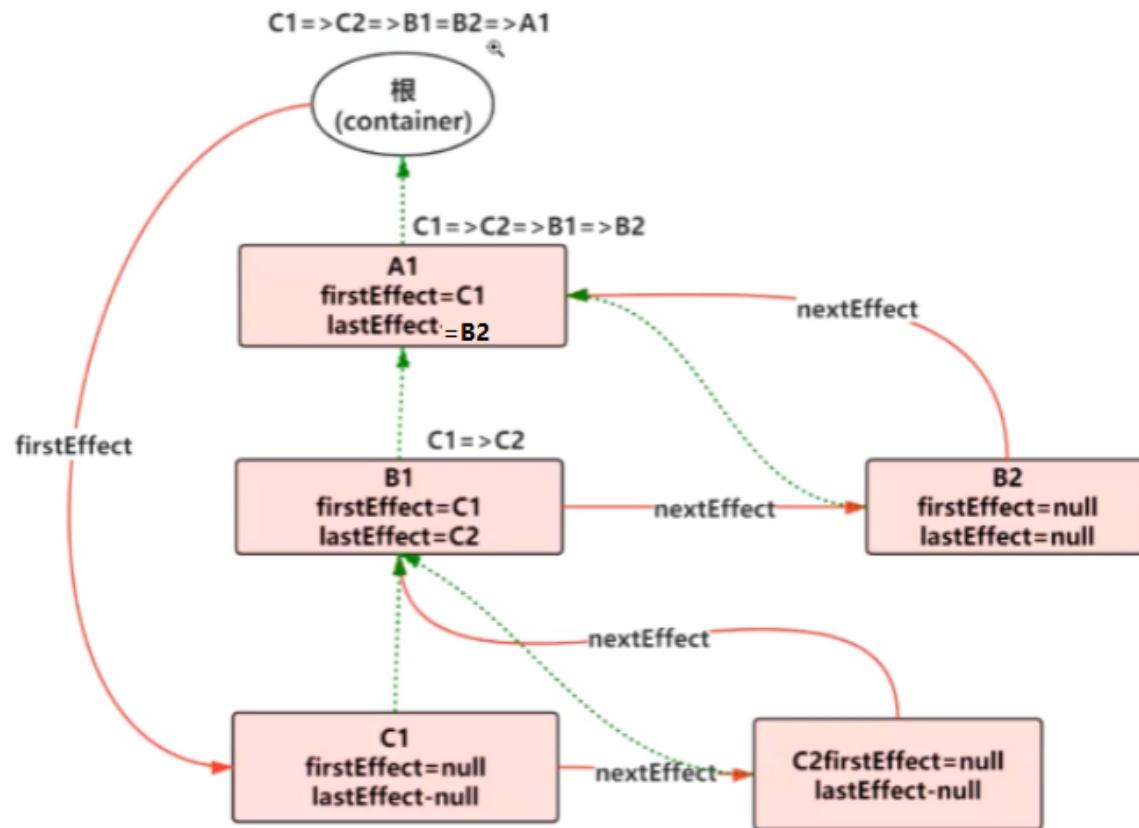
Fiber Tree 是由 Fiber Node 构成的，更像是一个单链表构成的树，便于向上/向下/向兄弟节点转换



4. `stateNode` 保存组件的实例。

副作用用种类和副作用链表相关：

- `effectTag`, 当需要变化的时候, 具体需要执行的操作的类型, 比如 `Update`, `Placement`。
- `nextEffect`, 下一个需要处理的有副作用的 Fiber。
- `firstEffect` 和 `lastEffect`, 本 Fiber 的子树中有副作用的第一个和最后一个 Fiber。



综上所述

`child` (孩子)、`return` (父) 和 `sibling` (兄弟)

`firstEffect` (副作用的第一个Fiber)、`lastEffect` (副作用的最后一个Fiber) 和 `nextEffect` (下一个需要处理的有副作用的 Fiber)

Fiber由两个链表组成

第一个是把虚拟Dom转换成Fiber链表的过程

第二个就是一个归并的过程将Fiber链表转换成Effectlist链表转成真实DOM渲染到页面的过程

更新相关的:

- `updateQueue`, 更新队列, 用于记录状态更新, 回调函数, DOM 更新的队列。
- `memoizedState`, 上一次更新 fiber 的 state。
- `memoizedProps`, 上一次更新 fiber 的 props。
- `pendingProps`, 新的 props, 将用于子组件或 DOM 元素的 props。

剩余执行时间相关的:

- `expirationTime`, 一个任务单元的可执行时间。
- `childExpirationTime`, 用来判断子树是否还有待完成的修改。

实现Fiber

```

import React from 'react'
import ReactDOM from 'react-dom'

```

```

# jsx => 虚拟dom ==> Fiber
let element = (
  <div id="A">
    <div id="B1">
      <div id="C1"></div>
      <div id="C2"></div>
    </div>
    <div id="B2"></div>
  </div>
)

# 根节点
let rootDom = document.getElementById('root')

const TAG_ROOT = 0
const TAG_PLACEMENT = 5

# 工作进程
let workInProgressFiber

# 根Fiber
let rootFiberNode = {
  # 标识
  tag: TAG_ROOT,
  # 对应的真实的dom是谁
  stateNode: rootDom,
  # 属性
  props: { children: [element] }
}

#就是让工作进程去循环根
workInProgressFiber = rootFiberNode

# 工作循环
#deadLine它是requestIdleCallback提供的得到空闲时间的形参对象
function workLoop(deadLine) {
  while (workInProgressFiber && deadLine.timeRemaining() > 0) {
    workInProgressFiber = performUnitOfWork(workInProgressFiber)
  }
  console.log(rootFiberNode)
}

# 进行转换 -- 深度优先
function performUnitOfWork(fiberNode) {
  beginWork(fiberNode)
  # 父有子
  if (fiberNode.child) {
    return fiberNode.child
  }
  # 没有子,一层层向上找
  while (fiberNode) {
    # 没有儿子
    completeEffect(fiberNode)

    # 看看兄弟
    if (fiberNode.sibling) {
      return fiberNode.sibling
    }
  }
}

```

```
}

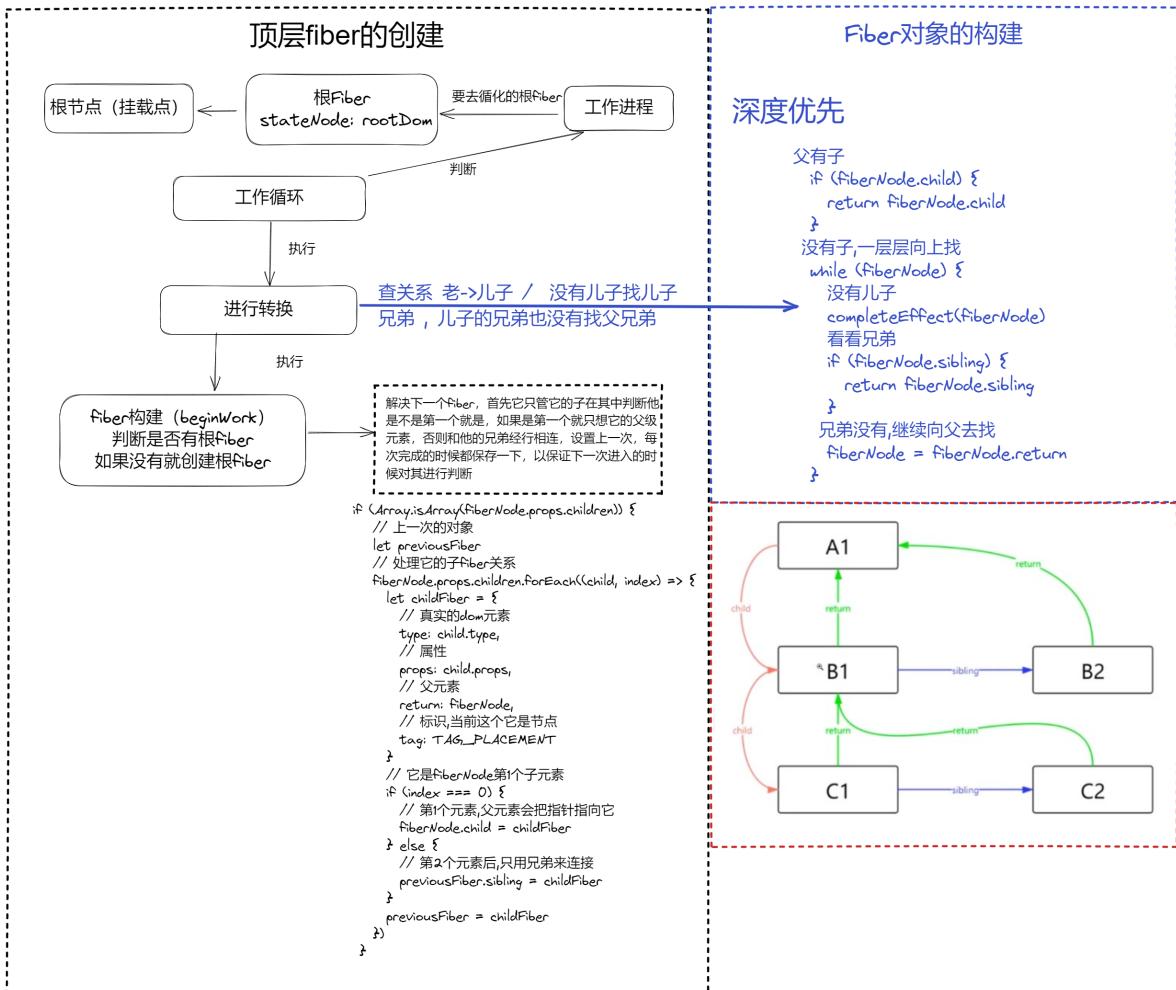
# 兄弟没有,继续向父去找
fiberNode = fiberNode.return
}

}

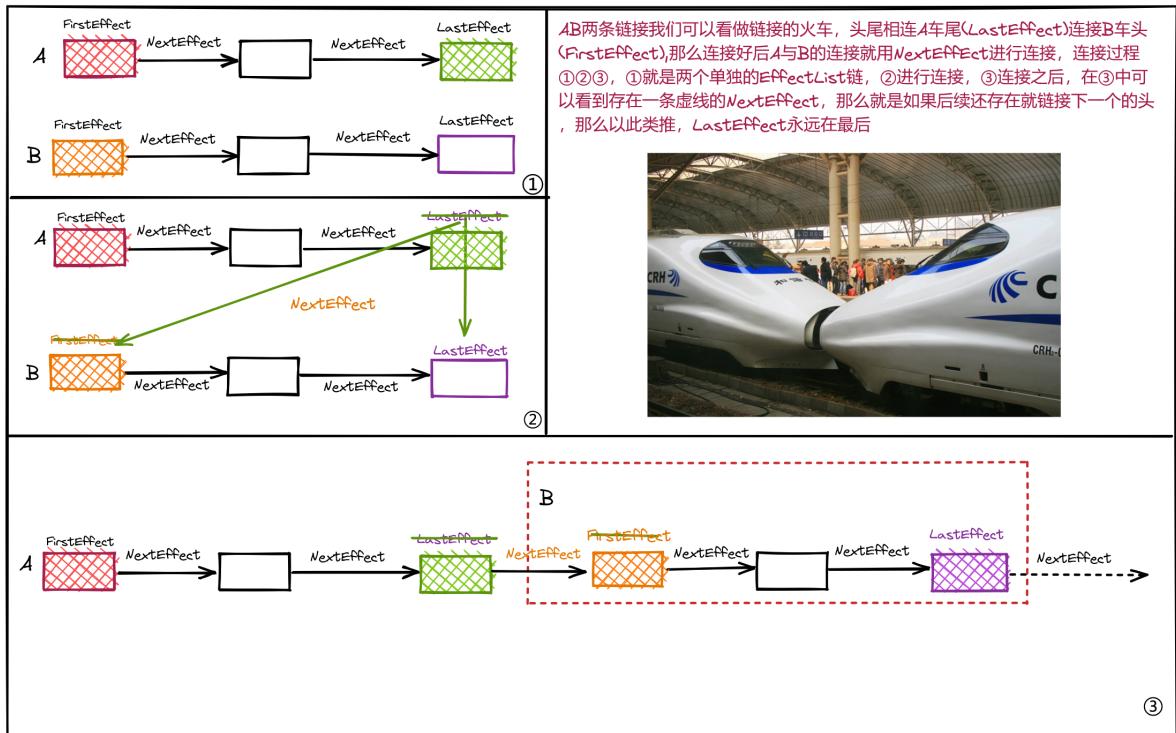
# fiber构建
function beginwork(fiberNode) {
# 当前fiber节点中stateNode不是一个对象时
if (!fiberNode.stateNode) {
# fiber中的type属性对应着真实的dom元素标签
fiberNode.stateNode = document.createElement(fiberNode.type)
}

# 解决下一个fiber问题
if (Array.isArray(fiberNode.props.children)) {#它有孩子, 只管第一个元素
# 上一次的对象
let previousFiber
# 处理它的子fiber关系
fiberNode.props.children.forEach((child, index) => {
let childFiber = {
# 真实的dom元素
type: child.type,
# 属性
props: child.props,
# 父元素
return: fiberNode,
# 标识,当前这个它是节点
tag: TAG_PLACEMENT
}
# 它是fiberNode第1个子元素
if (index === 0) {
# 第1个元素,父元素会把指针指向它
fiberNode.child = childFiber
} else {
# 第2个元素后,只用兄弟来连接
previousFiber.sibling = childFiber
}
previousFiber = childFiber
})
}
}

# 空余时间
window.requestIdleCallback(workLoop)
```



Effectlist (副作用链) 过程



```

import React from 'react'
import ReactDOM from 'react-dom'

// jsx => 虚拟dom ==> Fiber
let element =

```

```
<div id="A">
  <div id="B1">
    <div id="C1"></div>
    <div id="C2"></div>
  </div>
  <div id="B2"></div>
</div>
)
// console.log(element)
// 根节点
let rootDom = document.getElementById('root')

const TAG_ROOT = 0
const TAG_PLACEMENT = 5

// 工作进程
let workInProgressFiber

// 根Fiber
let rootFiberNode = {
  // 标识
  tag: TAG_ROOT,
  // 对应的真实的dom是谁
  stateNode: rootDom,
  // 属性
  props: { children: [element] }
}

workInProgressFiber = rootFiberNode

// 工作循环
function workLoop(deadLine) {
  while (workInProgressFiber && deadLine.timeRemaining() > 0) {
    workInProgressFiber = performUnitOfWork(workInProgressFiber)
  }
  // 把fiber生成为dom
  commitRoot()
}

// 把fiber生成为dom
function commitRoot() {
  let currentFiber = rootFiberNode.firstEffect
  while (currentFiber) {
    currentFiber.return.stateNode.appendChild(currentFiber.stateNode)
    // 向下渲染 直接为null
    currentFiber = currentFiber.nextEffect
  }
}

// 进行转换 -- 深度优先
// 查关系 老->儿子 / 没有儿子找儿子兄弟， 儿子的兄弟也没有找父兄弟
function performUnitOfWork(fiberNode) {
  beginWork(fiberNode)
  // 父有子
  if (fiberNode.child) {
    return fiberNode.child
  }
  // 没有子，一层层向上找
```

```

while (fiberNode) {
  // 没有儿子
  completeEffect(fiberNode)

  // 看看兄弟
  if (fiberNode.sibling) {
    return fiberNode.sibling
  }
  // 兄弟没有,继续向父去找
  fiberNode = fiberNode.return
}

// fiber构建
function beginwork(fiberNode) {
  // console.log('beginwork', fiberNode)
  // 当前fiber节点中stateNode不是一个对象时
  if (!fiberNode.stateNode) {
    // fiber中的type属性对应着真实的dom元素标签
    fiberNode.stateNode = document.createElement(fiberNode.type)
  }
  // 解决下一次fiber问题
  if (Array.isArray(fiberNode.props.children)) {
    // 上一次的对象
    let previousFiber
    // 处理它的子fiber关系
    fiberNode.props.children.forEach((child, index) => {
      let childFiber = {
        // 真实的dom元素
        type: child.type,
        // 属性
        props: child.props,
        // 父元素
        return: fiberNode,
        // 标识,当前这个它是节点
        tag: TAG_PLACEMENT
      }
      // 它是fiberNode第1个子元素
      if (index === 0) {
        // 第1个元素,父元素会把指针指向它
        fiberNode.child = childFiber
      } else {
        // 第2个元素后,只用兄弟来连接
        previousFiber.sibling = childFiber
      }
      previousFiber = childFiber
    })
  }
}

# 构建EffectList链表 为了生成dom所用
function completeEffect(fiberNode) {
  console.log('completeEffect', fiberNode)
  # 找父Fiber
  let returnFiber = fiberNode.return
  # 有父元素
  if (returnFiber) {
    if (!returnFiber.firstEffect) {

```

```

        returnFiber.firstEffect = fiberNode.firstEffect
    }

    if (fiberNode.lastEffect) {
        if (returnFiber.lastEffect) {
            returnFiber.lastEffect.nextEffect = fiberNode.firstEffect
        }
        returnFiber.lastEffect = fiberNode.lastEffect
    }

    # tag它是一个数字,所以你的判断是更换
    if (fiberNode.tag >= 0) {
        if (returnFiber.lastEffect) {
            returnFiber.lastEffect.nextEffect = fiberNode
        } else {
            returnFiber.firstEffect = fiberNode
        }
    }

    # 可创建的节点
    returnFiber.lastEffect = fiberNode
}

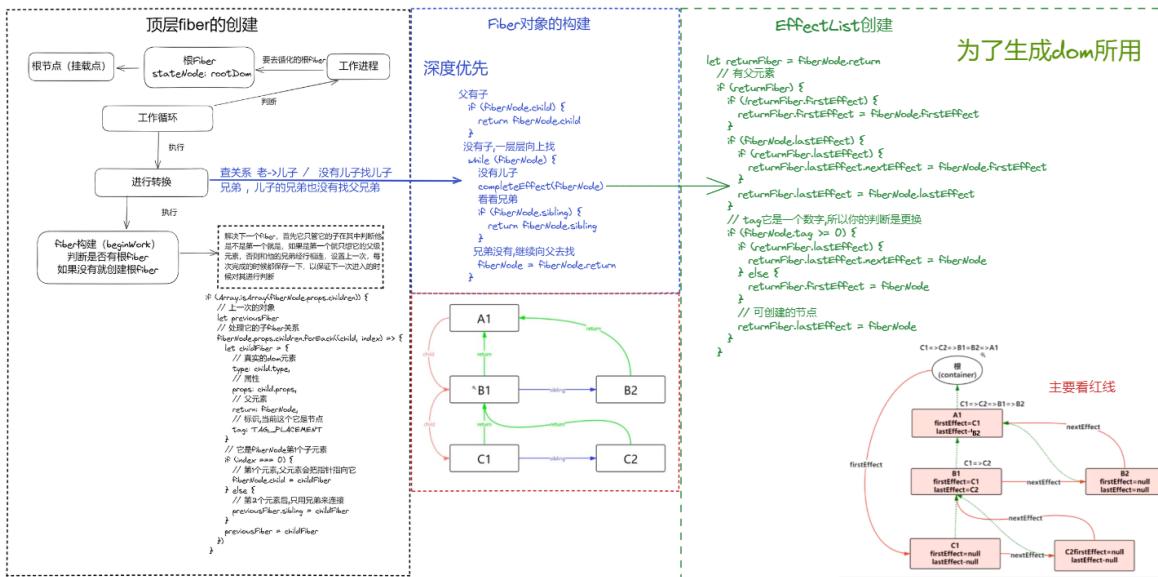
}

}

}

window.requestIdleCallback(workLoop)

```



把Fiber生成为DOM

```

import React from 'react'
import ReactDOM from 'react-dom'

// jsx => 虚拟dom ==> Fiber
let element = (
  <div id="A">
    <div id="B1">
      <div id="C1"></div>
      <div id="C2"></div>
    </div>
    <div id="B2"></div>
  </div>
)

// console.log(element)
// 根节点

```

```
let rootDom = document.getElementById('root')

const TAG_ROOT = 0
const TAG_PLACEMENT = 5

// 工作进程
let workInProgressFiber

// 根Fiber
let rootFiberNode = {
  // 标识
  tag: TAG_ROOT,
  // 对应的真实的dom是谁
  stateNode: rootDom,
  // 属性
  props: { children: [element] }
}

workInProgressFiber = rootFiberNode

// 工作循环
function workLoop(deadLine) {
  while (workInProgressFiber && deadLine.timeRemaining() > 0) {
    workInProgressFiber = performUnitOfWork(workInProgressFiber)
  }
  # 把fiber生成为dom
  commitRoot()
}

# 把fiber生成为dom
function commitRoot() {
  let currentFiber = rootFiberNode.firstEffect
  while (currentFiber) {
    currentFiber.return.stateNode.appendChild(currentFiber.stateNode)
    # 向下渲染 直接为null
    currentFiber = currentFiber.nextEffect
  }
}

// 进行转换 -- 深度优先
// 查关系 老->儿子 / 没有儿子找儿了兄弟， 儿子的兄弟也没有找父兄弟
function performUnitOfWork(fiberNode) {
  beginWork(fiberNode)
  // 父有子
  if (fiberNode.child) {
    return fiberNode.child
  }
  // 没有子,一层层向上找
  while (fiberNode) {
    // 没有儿子
    completeEffect(fiberNode)

    // 看看兄弟
    if (fiberNode.sibling) {
      return fiberNode.sibling
    }
    // 兄弟没有,继续向父去找
    fiberNode = fiberNode.return
  }
}
```

```

        }

    // fiber构建
    function beginwork(fiberNode) {
        // console.log('beginwork', fiberNode)
        // 当前fiber节点中stateNode不是一个对象时
        if (!fiberNode.stateNode) {
            // fiber中的type属性对应着真实的dom元素标签
            fiberNode.stateNode = document.createElement(fiberNode.type)
        }
        // 解决下一次fiber问题
        if (Array.isArray(fiberNode.props.children)) {
            // 上一次的对象
            let previousFiber
            // 处理它的子fiber关系
            fiberNode.props.children.forEach((child, index) => {
                let childFiber = {
                    // 真实的dom元素
                    type: child.type,
                    // 属性
                    props: child.props,
                    // 父元素
                    return: fiberNode,
                    // 标识,当前这个它是节点
                    tag: TAG_PLACEMENT
                }
                // 它是fiberNode第1个子元素
                if (index === 0) {
                    // 第1个元素,父元素会把指针指向它
                    fiberNode.child = childFiber
                } else {
                    // 第2个元素后,只用兄弟来连接
                    previousFiber.sibling = childFiber
                }
                previousFiber = childFiber
            })
        }
    }

    // 构建EffectList链表 为了生成dom所用
    function completeEffect(fiberNode) {
        console.log('completeEffect', fiberNode)
        // 找父Fiber
        let returnFiber = fiberNode.return
        // 有父元素
        if (returnFiber) {
            if (!returnFiber.firstEffect) {
                returnFiber.firstEffect = fiberNode.firstEffect
            }
            if (fiberNode.lastEffect) {
                if (returnFiber.lastEffect) {
                    returnFiber.lastEffect.nextEffect = fiberNode.firstEffect
                }
                returnFiber.lastEffect = fiberNode.lastEffect
            }
            // tag它是一个数字,所以你的判断是更换
            if (fiberNode.tag >= 0) {

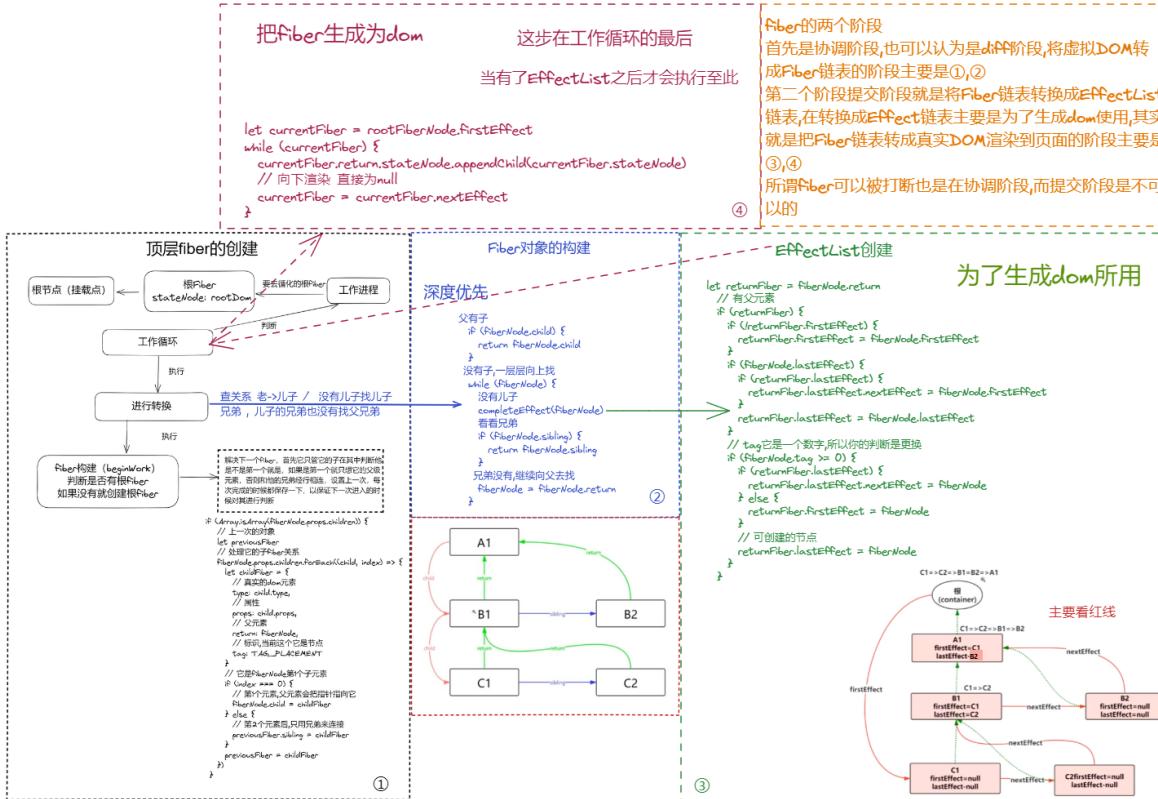
```

```

    if (returnFiber.lastEffect) {
      returnFiber.lastEffect.nextEffect = fiberNode
    } else {
      returnFiber.firstEffect = fiberNode
    }
    // 可创建的节点
    returnFiber.lastEffect = fiberNode
  }
}

window.requestIdleCallback(workLoop)

```



理解Fiber

如何理解fiber?

react尚存在一个明显的缺陷，这是大多数vDom框架都需要面对的 diff。

要知道，`render`前后的两颗 vDom tree 进行 diff，这个过程是不可中断的(以tree为单位，不可中断)，这将造成当diff的两颗tree足够庞大的时候，js线程会被diff阻塞，无法对并发事件作出回应。

为了解决这个问题

react将vDom节点上添加了链表节点的特性(就是所谓的fiber结构)，将其改造成了fiber节点(其实这就是vdom节点结合了链表节点的特性)，目的是为了后面的Fiber架构的实现，以实现应对并发事件的“并发模式”。简单来说就是时间分片，任务分炼。fiber节点的改造是为了将diff的单位从不可中断的tree降级到可中断的单一vDom。这意味着每次一个vDom节点比对过后，可以轮训是否发生了优先级更高的并发事件(react将用户想更快得到反馈的事进行了优先级排序，比如js动画等)。每个vDom的比对是一个时间片单位，因此基于fiber节点的Fiber架构，存在这样的特性：“时间分片，任务分炼，异步渲染”。

