link: null
title: 珠峰架构师成长计划
description: index.js
keywords: null
author: null
date: null
publisher: 珠峰架构师成长计划
stats: paragraph=61 sentences=176, words=1364

1. Fiber之前的React

index.js

```
import React from 'react';
import ReactDOM from 'react-dom';
let element = (
    <div id="A1">
        <div id="B1">
            <div id="C1">div>
            <div id="C2">div>
        div>
        <div id="B2">div>
    div>
)
console.log(JSON.stringify(element,null,2));
ReactDOM.render(element,document.getElementById('root'));
```

```
let element = {
    "type": "div",
    "key": "A1",
    "props": {
        "id": "A1",
        "children": [
            {
                "type": "div",
                "key": "B1",
                "props": {
                    "id": "B1",
                    "children": [
                        {
                            "type": "div",
                            "key": "C1",
                            "props": { "id": "C1"},
                        },
                        {
                            "type": "div",
                            "key": "C2",
                            "props": {"id": "C2"},
                        }
                    ]
                },
            },
            {
                "type": "div",
                "key": "B2",
                "props": {"id": "B2"},
            }
        ]
    },
}
function render(element, container) {
    let dom = document.createElement(element.type);
    Object.keys(element.props).filter(key => key !== 'children').forEach(key => {
        dom[key] = element.props[key];
    });
    if(Array.isArray(element.props.children)){
        element.props.children.forEach(child=>render(child,dom));
    }
    container.appendChild(dom);
}
render(element, document.getElementById('root'));
```

2. 帧

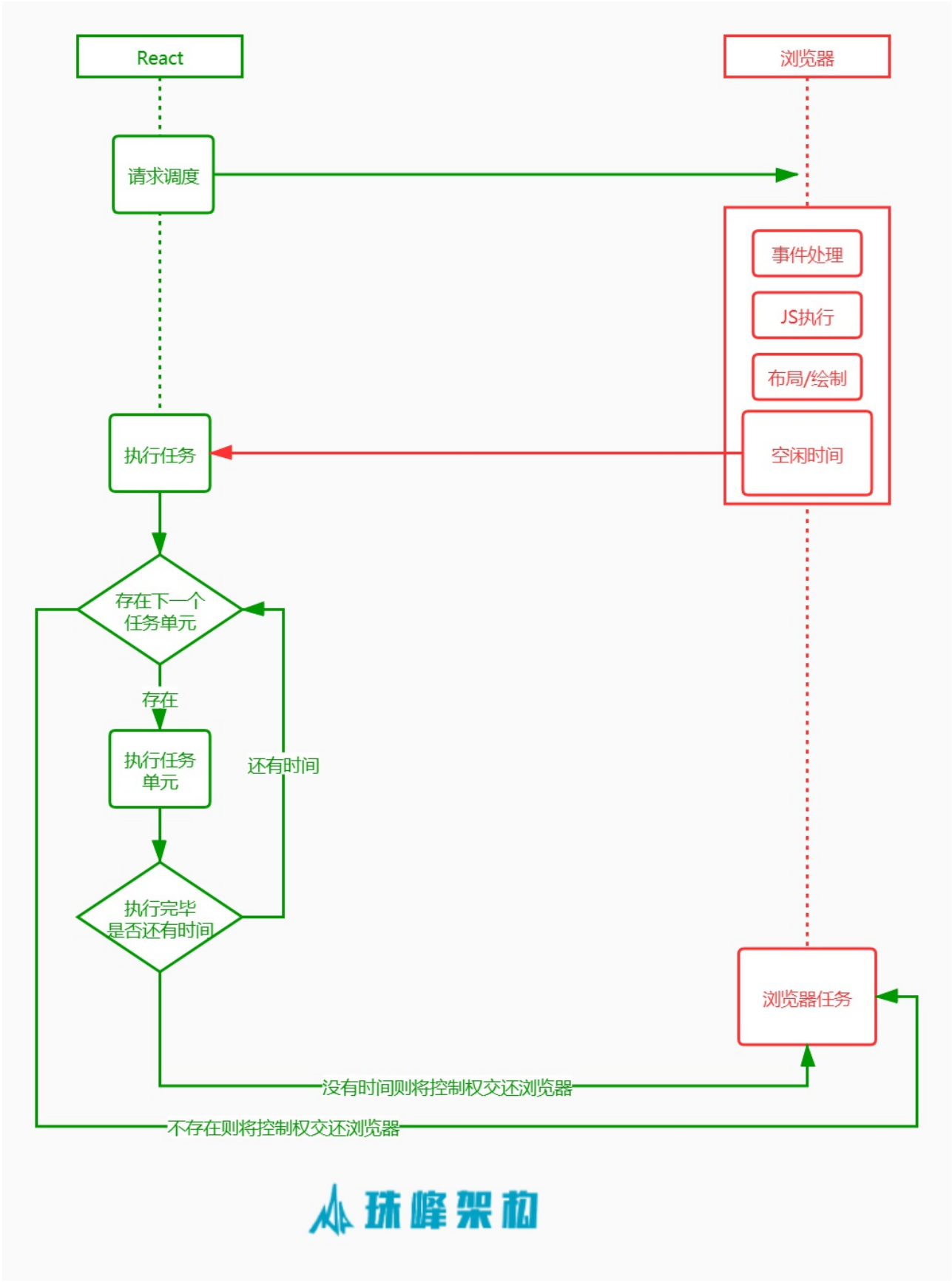
- 目前大多数设备的屏幕刷新率为 60 次/秒
- 当每秒绘制的帧数（FPS）达到 60 时，页面是流畅的,小于这个值时，用户会感觉到卡顿
- 每个帧的预算时间是16.66 毫秒 (1秒/60)
- 每个帧的开头包括样式计算、布局和绘制
- JavaScript执行 Javascript引擎和页面渲染引擎在同一个渲染线程,GUI渲染和Javascript执行两者是互斥的
- 如果某个任务执行时间过长，浏览器会推迟渲染

3. 什么是Fiber

- 我们可以通过某些调度策略合理分配CPU资源，从而提高用户的响应速度
- 通过 Fiber架构，让自己的协调过程变成可被中断。 适时地让出CPU执行权，除了可以让浏览器及时地响应用户的交互

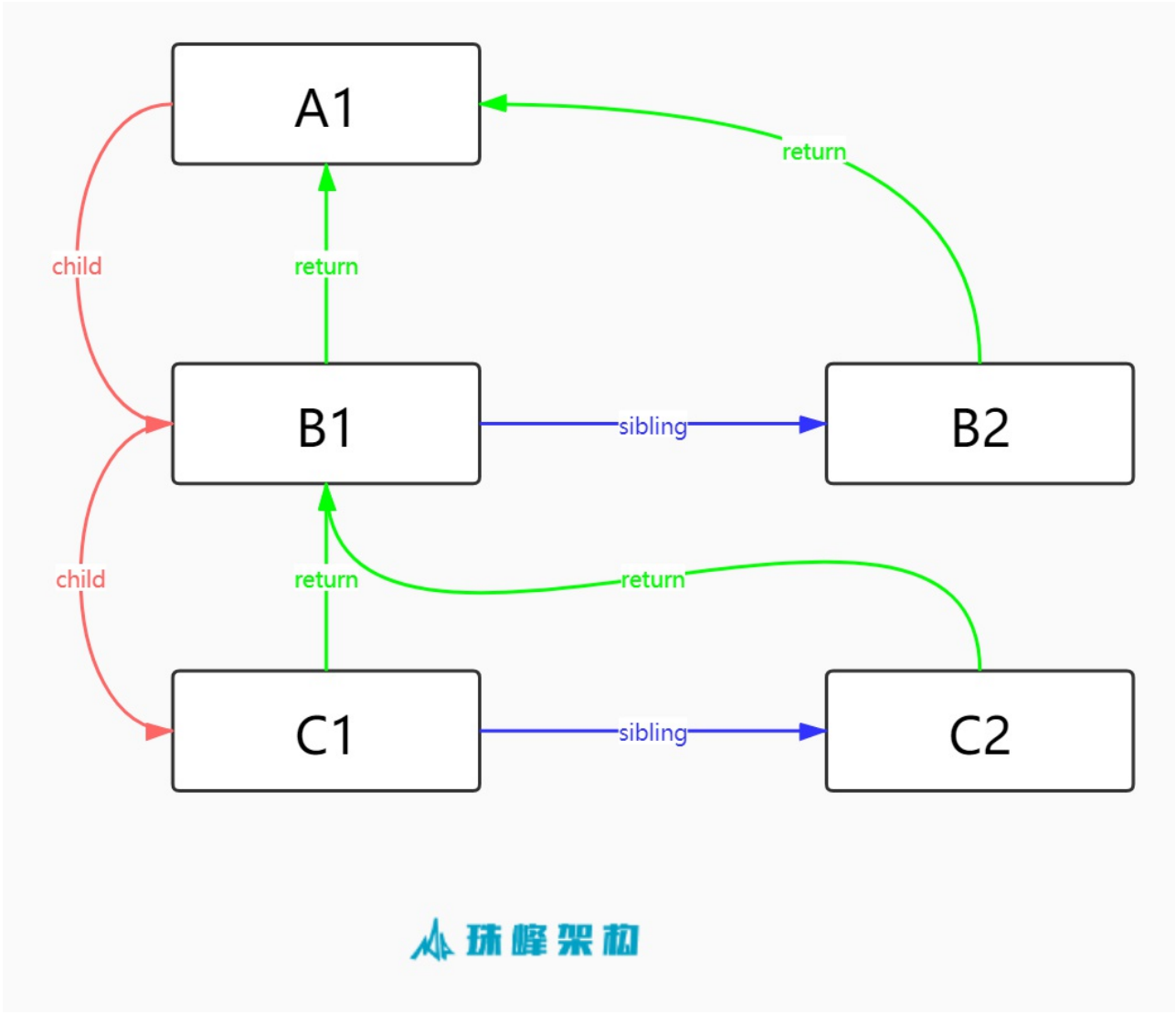
3.1 Fiber是一个执行单元

- Fiber是一个执行单元,每次执行完一个执行单元, React 就会检查现在还剩多少时间，如果没有时间就将控制权让出去



3.2 Fiber是一种数据结构 #

- React目前的做法是使用链表,每个虚拟节点内部表示为一个Fiber



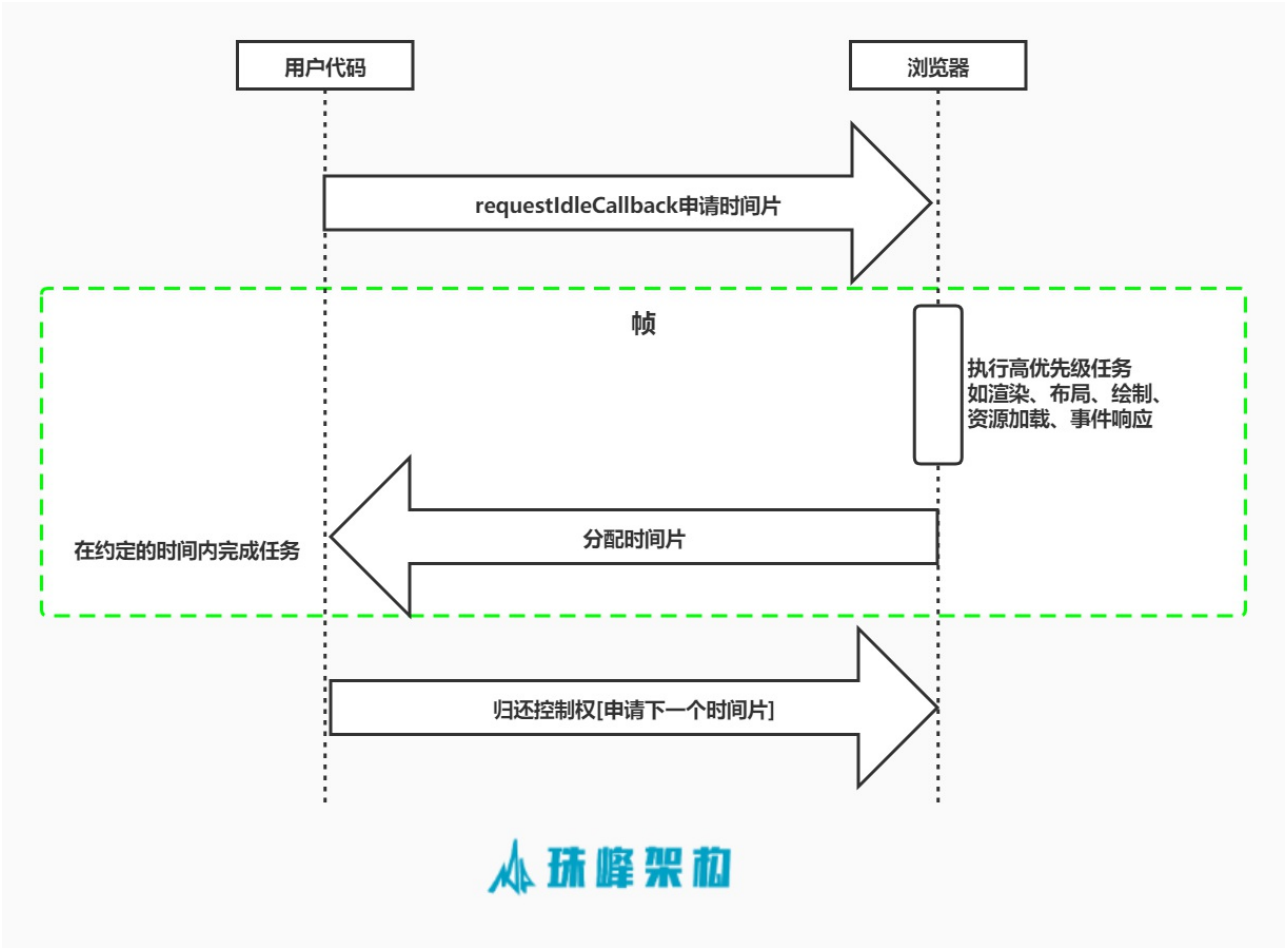
4.rAF

- requestAnimationFrame (https://developer.mozilla.org/zh-CN/docs/Web/API/Window/requestAnimationFrame)回调函数会在绘制之前执行
- `requestAnimationFrame(callback)` 会在浏览器每次重绘前执行 callback 回调, 每次 callback 执行的时机都是浏览器刷新下一帧渲染周期的起点上
- `requestAnimationFrame(callback)` 的回调 callback 回调参数 timestamp 是回调被调用的时间，也就是当前帧的起始时间
- `rAfTime performance.timing.navigationStart + performance.now() &#x7EA6;&#x7B49;&#x4E8E; Date.now()`

```
<html lang="en">

<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>RAFtitle>
head>

<body>
    <div style="background: lightblue;width: 0;height: 20px;">div>
    <button>开始button>
    <script>

        const div = document.querySelector('div');
        const button = document.querySelector('button');
        let start;
        function progress(rAfTime) {
            div.style.width = div.offsetWidth + 1 + 'px';
            div.innerHTML = (div.offsetWidth) + '%';
            if (div.offsetWidth < 100) {
                let current = Date.now();
                console.log((current - start)+'ms');
                start = current;
                timer = requestAnimationFrame(progress);
            }
        }
        button.onclick = () => {
            div.style.width = 0;
            start = Date.now();
            requestAnimationFrame(progress);
        }
    script>
body>
html>
```

5.requestIdleCallback

- 我们希望快速响应用户，让用户觉得够快，不能阻塞用户的交互
- requestIdleCallback (https://developer.mozilla.org/zh-CN/docs/Web/API/Window/requestIdleCallback)使开发者能够在主事件循环上执行后台和低优先级工作，而不会影响延迟关键事件，如动画和输入响应
- 正常帧任务完成后没超过16 ms,说明时间有富余，此时就会执行 requestIdleCallback 里注册的任务
- requestAnimationFrame的回调会在每一帧确定执行，属于高优先级任务，而 requestIdleCallback的回调则不一定，属于低优先级任务




```
window.requestIdleCallback(
  callback: (deaLine: IdleDeadline) => void,
  option?: {timeout: number}
)

interface IdleDeadline {
  didTimeout: boolean
  timeRemaining(): DOMHighResTimeStamp
}
```

- callback：回调即空闲时需要执行的任务，该回调函数接收一个IdleDeadline对象作为入参。其中IdleDeadline对象包含：
  - didTimeout，布尔值，表示任务是否超时，结合 timeRemaining 使用
  - timeRemaining()，表示当前帧剩余的时间，也可理解为留给任务的时间还有多少
- options：目前 options 只有一个参数
  - timeout。表示超过这个时间后，如果任务还没执行，则强制执行，不必等待空闲

```
<body>
    <script>
        function sleep(duration) {
           let start =Date.now();
           while(start+duration>Date.now()){}
        }
        const works = [
            () => {
                console.log("第1个任务开始");
                sleep(0);
                console.log("第1个任务结束");
            },
            () => {
                console.log("第2个任务开始");
                sleep(0);
                console.log("第2个任务结束");
            },
            () => {
                console.log("第3个任务开始");
                sleep(0);
                console.log("第3个任务结束");
            },
        ];

        requestIdleCallback(workLoop, { timeout: 1000 });
        function workLoop(deadline) {
            console.log('本帧剩余时间', parseInt(deadline.timeRemaining()));
            while ((deadline.timeRemaining() > 1 || deadline.didTimeout) && works.length > 0) {
                performUnitOfWork();
            }

            if (works.length > 0) {
                console.log(`只剩下${parseInt(deadline.timeRemaining())}ms,时间片到了等待下次空闲时间的调度`);
                requestIdleCallback(workLoop);
            }
        }
        function performUnitOfWork() {
            works.shift()();
        }
    script>
body>
```

6.MessageChannel

- 目前 requestIdleCallback 目前只有Chrome支持
- 所以目前 React利用 MessageChannel模拟了requestIdleCallback，将回调延迟到绘制操作之后执行
- MessageChannel API允许我们创建一个新的消息通道，并通过它的两个MessagePort属性发送数据
- MessageChannel创建了一个通信的管道，这个管道有两个端口，每个端口都可以通过postMessage发送数据，而一个端口只要绑定了onmessage回调方法，就可以接收从另一个端口传过来的数据
- MessageChannel是一个宏任务


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

```
var channel = new MessageChannel();
var port1 = channel.port1;
var port2 = channel.port2;
port1.onmessage = function(event) {
    console.log("port1收到来自port2的数据: " + event.data);
}
port2.onmessage = function(event) {
    console.log("port2收到来自port1的数据: " + event.data);
}
port1.postMessage("发送给port2");
port2.postMessage("发送给port1");
```

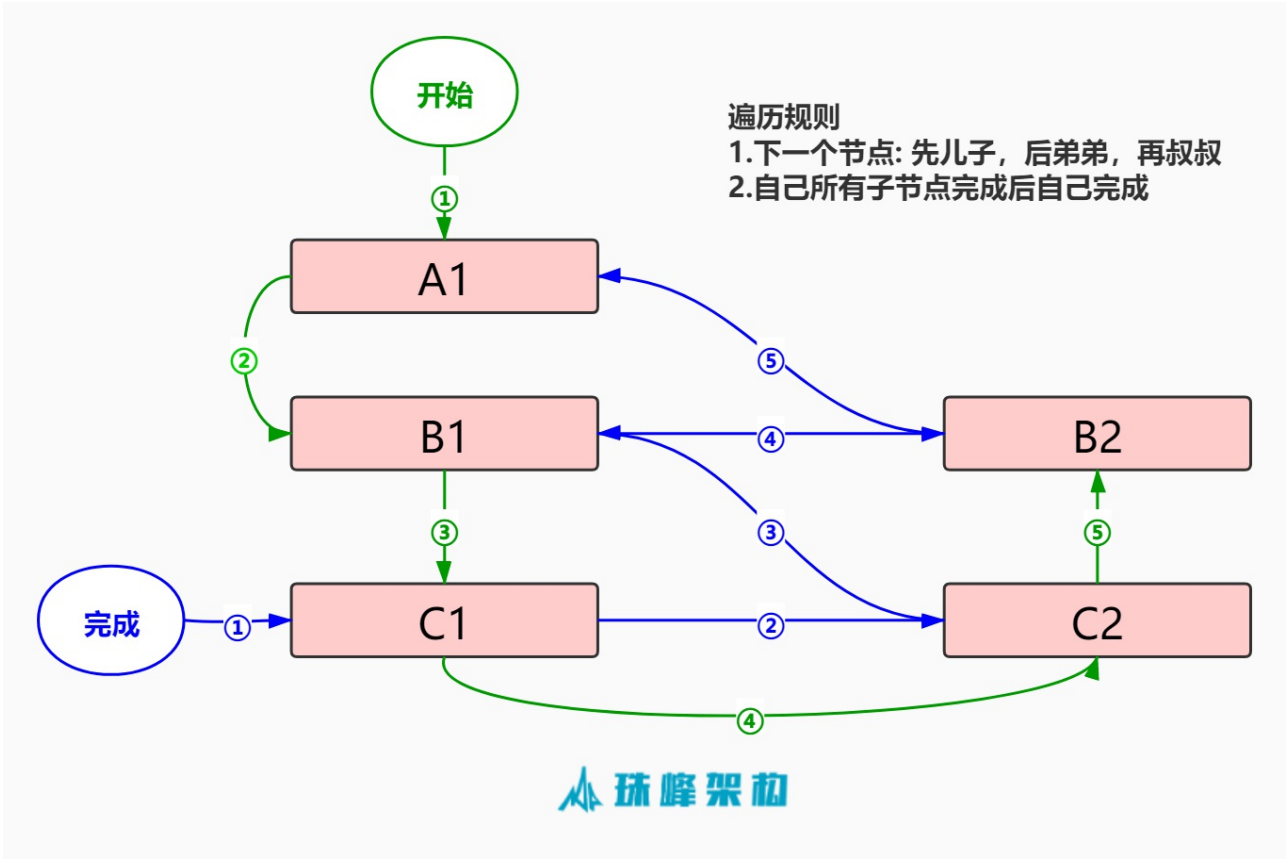
```
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Documenttitle>
head>
<body>
    <script>
        const channel = new MessageChannel()
        let pendingCallback;
        let activeFrameTime = (1000 / 60);

        let timeRemaining = () => frameDeadline - performance.now();
        channel.port2.onmessage = () => {
            var currentTime = performance.now();
            var didTimeout = frameDeadline  if(didTimeout || timeRemaining()>1){
                if (pendingCallback) {
                  pendingCallback({ didTimeout: frameDeadline  window.requestIdleCallback = (callback, options) => {
            requestAnimationFrame((rafTime) => {
                frameDeadline = rafTime + activeFrameTime;
                pendingCallback = callback;

                channel.port1.postMessage('hello');
            })
        }

        function sleep(d) {
            for (var t = Date.now(); Date.now() - t const works = [
            () => {
                console.log("第1个任务开始");
                sleep(20);
                console.log("第1个任务结束");
            },
            () => {
                console.log("第2个任务开始");
                sleep(20);
                console.log("第2个任务结束");
            },
            () => {
                console.log("第3个任务开始");
                sleep(0);
                console.log("第3个任务结束");
            },
        ];

        requestIdleCallback(workLoop, { timeout: 60 * 1000 });
        function workLoop(deadline) {
            console.log('本帧剩余时间', parseInt(deadline.timeRemaining()));
            while ((deadline.timeRemaining() > 1 || deadline.didTimeout) && works.length > 0) {
                performUnitOfWork();
            }
            if (works.length > 0) {
                console.log(`只剩下${parseInt(deadline.timeRemaining())}ms,时间片到了等待下次空闲时间的调度`);
                requestIdleCallback(workLoop, { timeout: 2 * 1000 });
            }
        }
        function performUnitOfWork() {
            works.shift()();
        }
    script>
body>

html>
```

7.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: 可以认为是 Diff 阶段, 这个阶段可以被中断, 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(Effect)
    - 提交阶段: 将上一个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

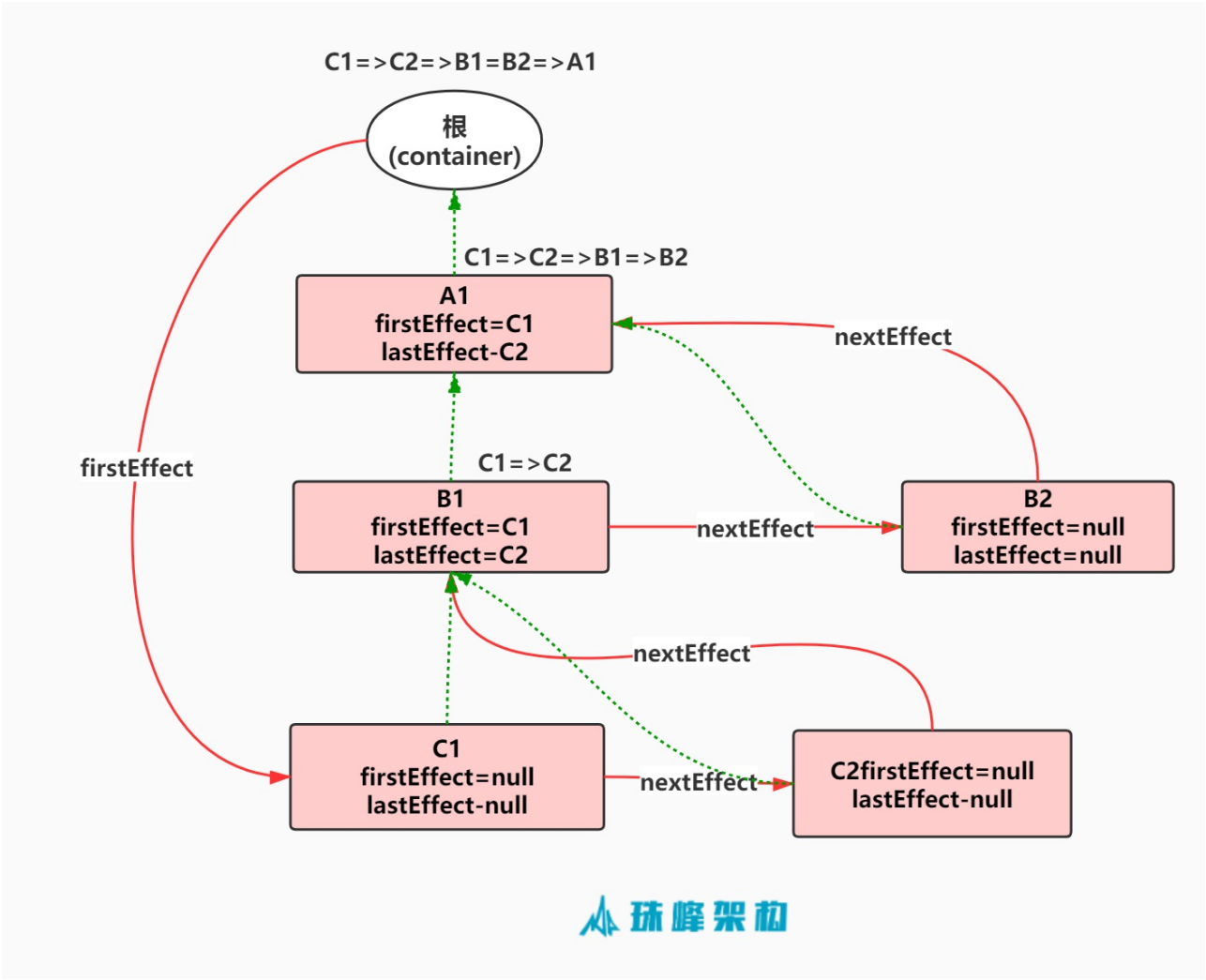
7.1 render阶段

- 从顶点开始遍历
- 如果有第一个儿子，先遍历第一个儿子
- 如果没有第一个儿子，标志着此节点遍历完成
- 如果有弟弟遍历弟弟
- 如果有没有下一个弟弟，返回父节点标识完成父节点遍历，如果有叔叔遍历叔叔
- 没有父节点遍历结束
- 先儿子，后弟弟，再叔叔,辈份越小越优先
- 什么时候一个节点遍历完成? 没有子节点，或者所有子节点都遍历完成了
- 没爹了就表示全部遍历完成了




```
let A1 = { type: 'div', props:{id: 'A1'} };
let B1 = { type: 'div', props:{id: 'B1'}, return: A1 };
let B2 = { type: 'div', props:{id: 'B2'}, return: A1 };
let C1 = { type: 'div', props:{id: 'C1'}, return: B1 };
let C2 = { type: 'div', props:{id: 'C2'}, return: B1 };
A1.child = B1;
B1.sibling = B2;
B1.child = C1;
C1.sibling = C2;

let nextUnitOfWork = null;

function workLoop() {
    while (nextUnitOfWork) {

        nextUnitOfWork = performUnitOfWork(nextUnitOfWork);
    }
    console.log('render阶段结束');

}
function performUnitOfWork(fiber) {
    let child = beginWork(fiber);
    if(child){
      return child;
    }
    while (fiber) {
        completeUnitOfWork(fiber);
        if (fiber.sibling) {
            return fiber.sibling;
        }
        fiber = fiber.return;
    }
}
function beginWork(fiber) {
    console.log('beginWork', fiber.props.id);
    return fiber.child;
}
function completeUnitOfWork(fiber) {
    console.log('completeUnitOfWork', fiber.props.id);
}
nextUnitOfWork = A1;
workLoop();
```

7.2 commit阶段


C1=>C2=>B1=B2=>A1
根
(container)
C1=>C2=>B1=>B2
A1
firstEffect=C1
lastEffect-C2
nextEffect
firstEffect
C1=>C2
B1
firstEffect=C1
lastEffect=C2
nextEffect
B2
firstEffect=null
lastEffect=null
nextEffect
C1
firstEffect=null
lastEffect-null
nextEffect
C2firstEffect=null
lastEffect-null
珠峰架构

```
let container = document.getElementById('root');
let C1 = { type: 'div', props: { id: 'C1', children: [] } };
let C2 = { type: 'div', props: { id: 'C2', children: [] } };
let B1 = { type: 'div', props: { id: 'B1', children: [C1, C2] } };
let B2 = { type: 'div', props: { id: 'B2', children: [] } };
let A1 = { type: 'div', props: { id: 'A1', children: [B1, B2] } };

let nextUnitOfWork = null;
let workInProgressRoot = null;
function workLoop() {
    while (nextUnitOfWork) {
        nextUnitOfWork = performUnitOfWork(nextUnitOfWork);
    }
    if (!nextUnitOfWork) {
        commitRoot();
    }
}
function commitRoot() {
    let fiber = workInProgressRoot.firstEffect;
    while (fiber) {
        console.log(fiber.props.id);
        commitWork(fiber);
        fiber = fiber.nextEffect;
    }
    workInProgressRoot = null;
}
function commitWork(currentFiber) {
    currentFiber.return.stateNode.appendChild(currentFiber.stateNode);
}
function performUnitOfWork(fiber) {
    beginWork(fiber);
    if (fiber.child) {
        return fiber.child;
    }
    while (fiber) {
        completeUnitOfWork(fiber);
        if (fiber.sibling) {
            return fiber.sibling;
        }
        fiber = fiber.return;
    }
}
function beginWork(currentFiber) {
    if (!currentFiber.stateNode) {
        currentFiber.stateNode = document.createElement(currentFiber.type);
        for (let key in currentFiber.props) {
            if (key !== 'children' && key !== 'key')
                currentFiber.stateNode[key]=currentFiber.props[key];
        }
    }
    let previousFiber;
    currentFiber.props.children.forEach((child, index) => {
        let childFiber = {
            type: child.type,
            props: child.props,
            return: currentFiber,
            effectTag: 'PLACEMENT',
            nextEffect: null
        }
        if (index === 0) {
            currentFiber.child = childFiber;
        } else {
            previousFiber.sibling = childFiber;
        }
        previousFiber = childFiber;
    });
}
function completeUnitOfWork(currentFiber) {
    const returnFiber = currentFiber.return;
    if (returnFiber) {
        if (!returnFiber.firstEffect) {
            returnFiber.firstEffect = currentFiber.firstEffect;
        }
        if (currentFiber.lastEffect) {
            if (returnFiber.lastEffect) {
                returnFiber.lastEffect.nextEffect = currentFiber.firstEffect;
            }
            returnFiber.lastEffect = currentFiber.lastEffect;
        }

        if (currentFiber.effectTag) {
            if (returnFiber.lastEffect) {
                returnFiber.lastEffect.nextEffect = currentFiber;
            } else {
                returnFiber.firstEffect = currentFiber;
            }
            returnFiber.lastEffect = currentFiber;
        }
    }
}

workInProgressRoot = {
    key: 'ROOT',
    stateNode: container,
    props: { children: [A1] }
};
nextUnitOfWork = workInProgressRoot;
workLoop();
```