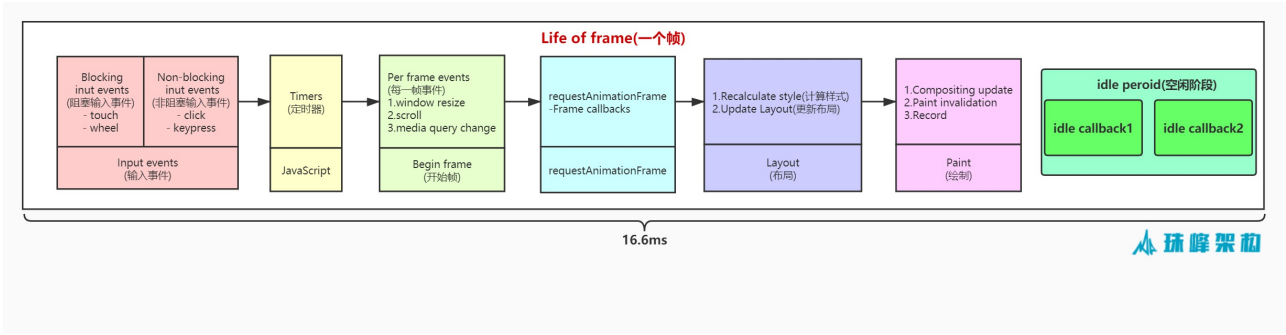
link: null
title: 珠峰架构师成长计划
description: null
keywords: null
author: null
date: null
publisher: 珠峰架构师成长计划
stats: paragraph=83 sentences=213, words=1527

1. 屏幕刷新率

- 目前大多数设备的屏幕刷新率为 60 次/秒
- 浏览器渲染动画或页面的每一帧的速率也需要跟设备屏幕的刷新率保持一致
- 页面是一帧一帧绘制出来的，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)
- 1s 60帧，所以每一帧分到的时间是 1000/60 ≈ 16 ms。所以我们书写代码时力求不让一帧的工作量超过 16ms

2. 帧

- 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



2.1 rAF#

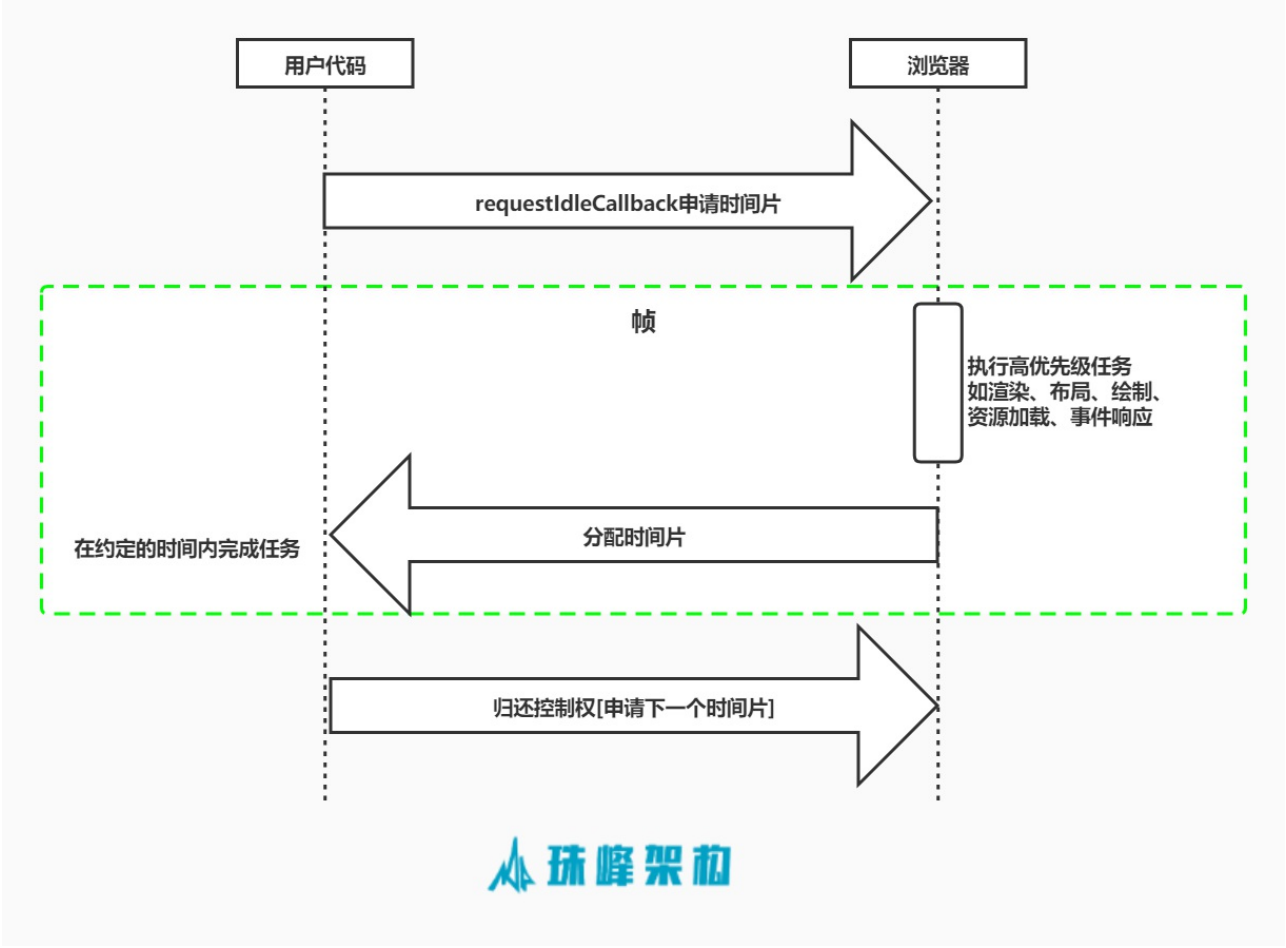
- requestAnimationFrame (https://developer.mozilla.org/zh-CN/docs/Web/API/Window/requestAnimationFrame)回调函数会在绘制之前执行

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

2.2 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




```
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(d) {
            for (var t = Date.now(); Date.now() - t 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>
```

2.3 MessageChannel

- 目前 requestIdleCallback 目前只有Chrome支持
- 所以目前 React利用MessageChannel (https://developer.mozilla.org/zh-CN/docs/Web/API/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 startTime;
        let timeoutTime;
        let perFrameTime = (1000 / 60);
        let timeRemaining = () => perFrameTime - (Date.now() - startTime);
        channel.port2.onmessage = () => {
            if (pendingCallback) {
                pendingCallback({ didTimeout: Date.now() > timeoutTime, timeRemaining });
            }
        }
        window.requestIdleCallback = (callback, options) => {
            timeoutTime = Date.now() + options.timeout;
            requestAnimationFrame(() => {
                startTime = Date.now();
                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(30);
                console.log("第1个任务结束");
            },
            () => {
                console.log("第2个任务开始");
                sleep(30);
                console.log("第2个任务结束");
            },
            () => {
                console.log("第3个任务开始");
                sleep(30);
                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: 60 * 1000 });
            }
        }
        function performUnitOfWork() {
            works.shift()();
        }
    script>
body>

html>
```

3. 单链表

- 单链表是一种链式存取的数据结构
- 链表中的数据是以节点来表示的，每个节点的构成：元素 + 指针(指示后继元素存储位置)，元素就是存储数据的存储单元，指针就是连接每个节点的地址

□

□

```
class Update {
    constructor(payload) {
        this.payload = payload;
        this.nextUpdate = null;
    }
}
class UpdateQueue {
    constructor() {
        this.baseState = null;
        this.firstUpdate = null;
        this.lastUpdate = null;
    }
    clear() {
        this.firstUpdate = null;
        this.lastUpdate = null;
    }
    enqueueUpdate(update) {
        if (this.firstUpdate === null) {
            this.firstUpdate = this.lastUpdate = update;
        } else {
            this.lastUpdate.nextUpdate = update;
            this.lastUpdate = update;
        }
    }
    forceUpdate() {
        let currentState = this.baseState || {};
        let currentUpdate = this.firstUpdate;
        while (currentUpdate) {
            let nexState = typeof currentUpdate.payload == 'function' ? currentUpdate.payload(currentState) : currentUpdate.payload;
            currentState = { ...currentState, ...nexState };
            currentUpdate = currentUpdate.nextUpdate;
        }
        this.firstUpdate = this.lastUpdate = null;
        this.baseState = currentState;
        return currentState;
    }
}

let queue = new UpdateQueue();
queue.enqueueUpdate(new Update({ name: 'zhufeng' }));
queue.enqueueUpdate(new Update({ number: 0 }));
queue.enqueueUpdate(new Update(state => ({ number: state.number + 1 })));
queue.enqueueUpdate(new Update(state => ({ number: state.number + 1 })));
queue.forceUpdate();
console.log(queue.baseState);
```

4.Fiber历史

4.1 Fiber之前的协调

- React 会递归比对VirtualDOM树，找出需要变动的节点，然后同步更新它们。这个过程 React 称为Reconcilation(协调)
- 在 Reconcilation期间，React 会一直占用着浏览器资源，一则会导致用户触发的事件得不到响应, 二则会导致掉帧，用户可能会感觉到卡顿

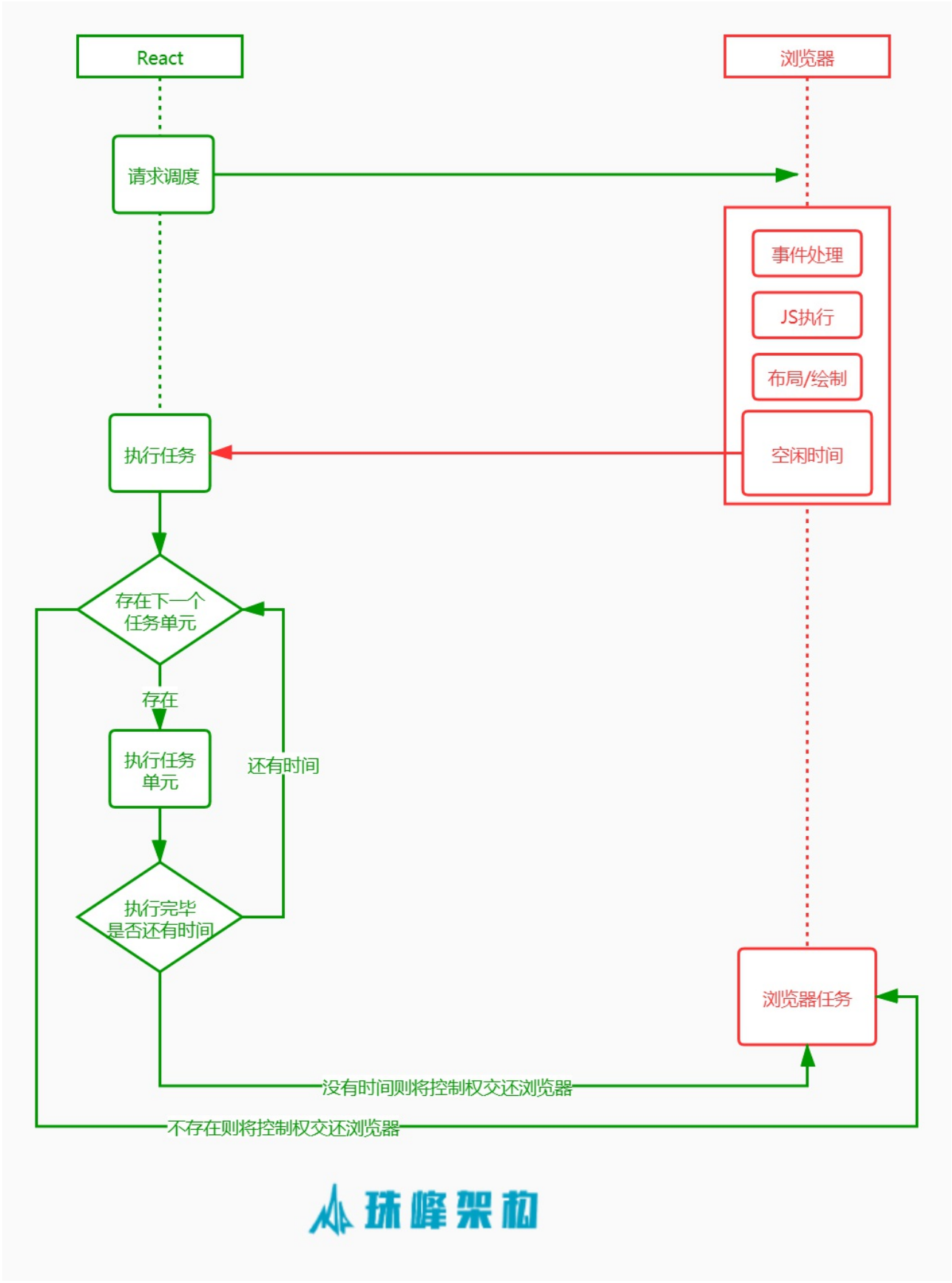
```
let root = {
    key: 'A1',
    children: [
        {
            key: 'B1',
            children: [
                {
                    key: 'C1',
                    children: []
                },
                {
                    key: 'C2',
                    children: []
                }
            ]
        },
        {
            key: 'B2',
            children: []
        }
    ]
}
function walk(element) {
    doWork(element);
    element.children.forEach(walk);
}

function doWork(element) {
    console.log(element.key);
}
walk(root);
```

4.2 Fiber是什么

- 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Reconcilation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

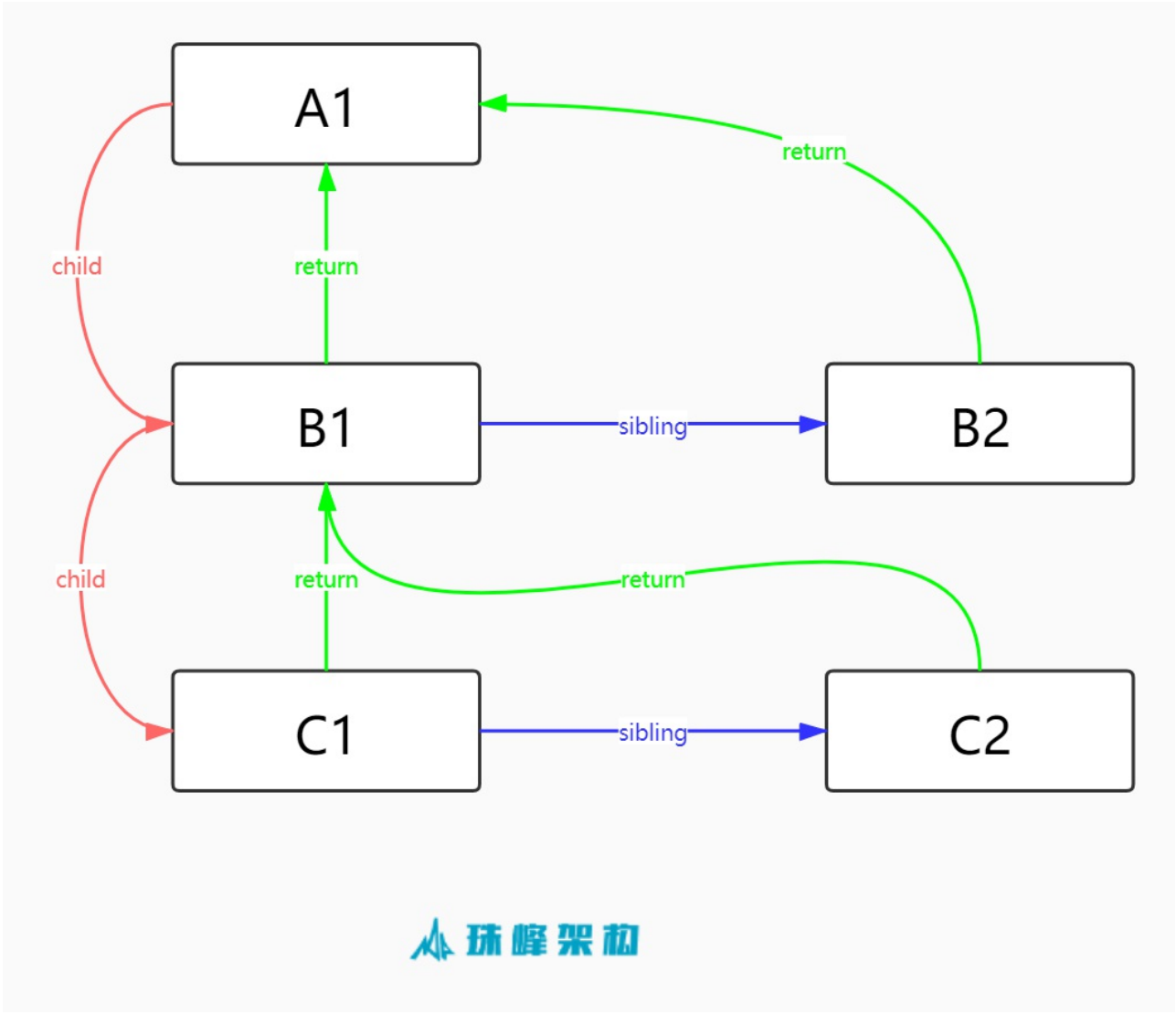
4.2.1 Fiber是一个执行单元

- 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



4.2.2 Fiber是一种数据结构

- React目前的做法是使用链表, 每个 VirtualDOM 节点内部表示为一个 `Fiber`




```
type Fiber = {

  type: any,

  return: Fiber,

  child: Fiber,

  sibling: Fiber
}
```

5.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

5.1 render阶段

- render阶段会构建fiber树

5.1.1 element.js

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

5.1.2 render.js

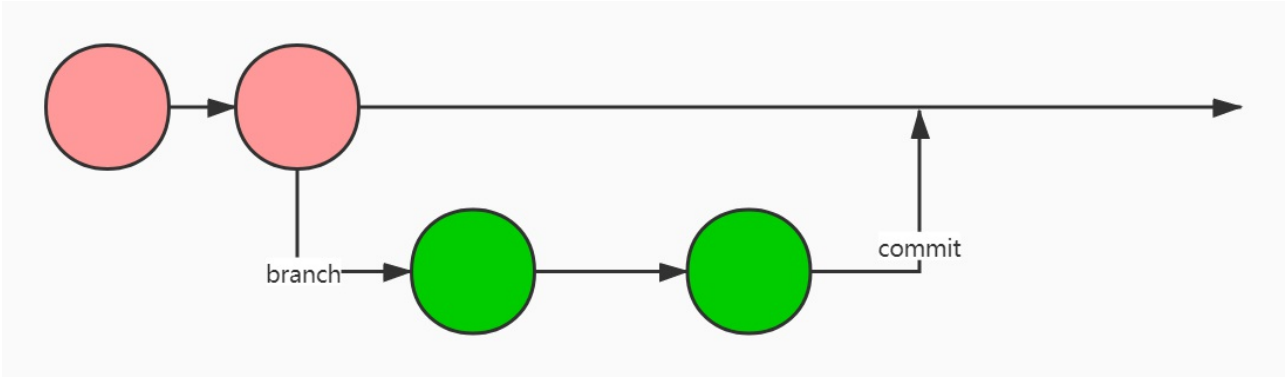
- 从顶点开始遍历
- 如果有第一个儿子，先遍历第一个儿子
- 如果没有第一个儿子，标志着此节点遍历完成
- 如果有弟弟遍历弟弟
- 如果有没有下一个弟弟，返回父节点标识完成父节点遍历，如果有叔叔遍历叔叔
- 没有父节点遍历结束

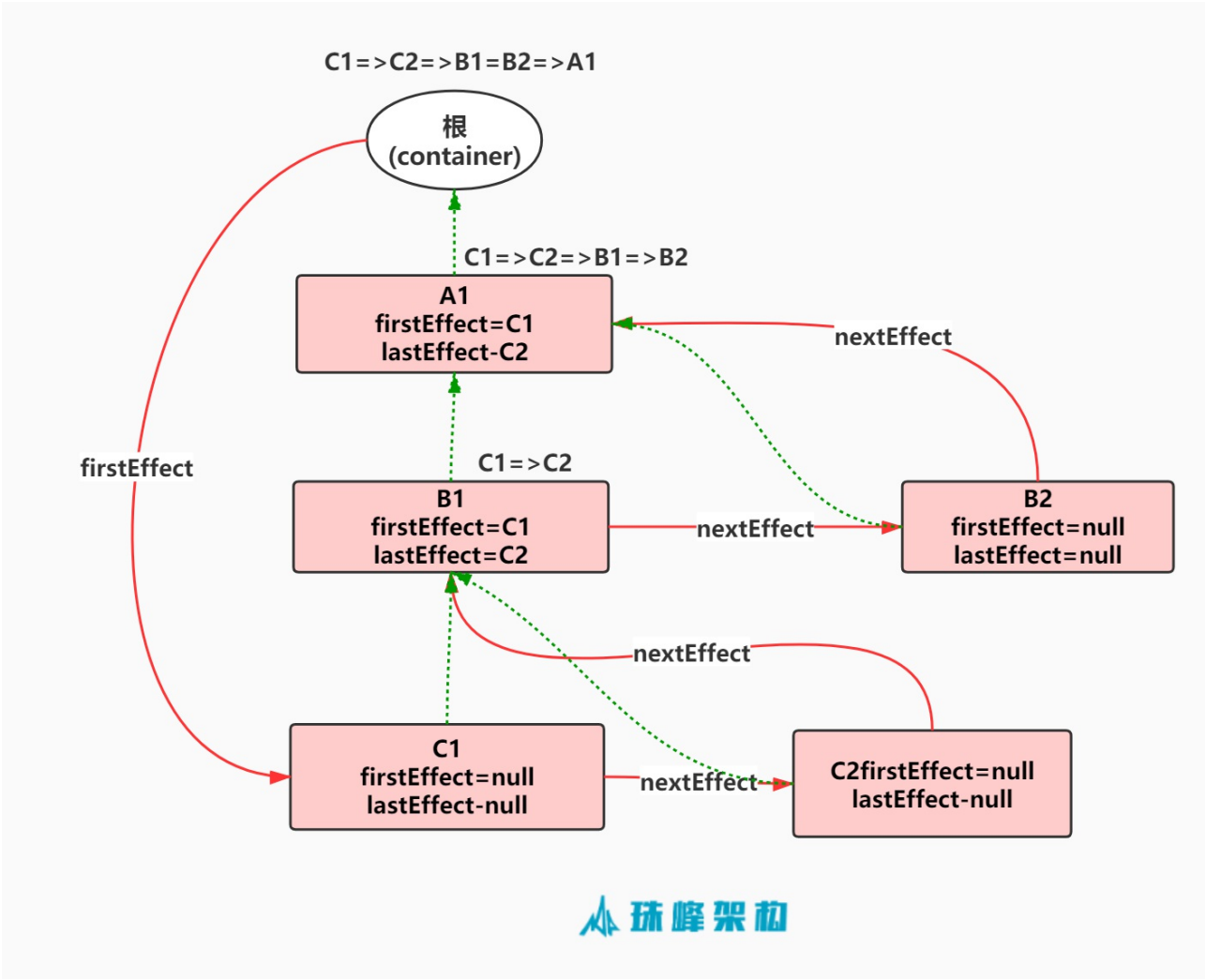
```
let rootFiber = require('./element');

let nextUnitOfWork = null;

function workLoop() {
    while (nextUnitOfWork) {

        nextUnitOfWork = performUnitOfWork(nextUnitOfWork);
    }

}
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(fiber) {
    console.log('beginWork', fiber.key);

}
function completeUnitOfWork(fiber) {
    console.log('completeUnitOfWork', fiber.key);
}
nextUnitOfWork = rootFiber;
workLoop();
```

- 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

5.2 commit阶段

- 类比Git分支功能,从旧树中fork出来一份，在新分支进行添加、删除和更新操作，经过测试后进行提交




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', key: 'C1', props: { id: 'C1', children: [] } };
let C2 = { type: 'div', key: 'C2', props: { id: 'C2', children: [] } };
let B1 = { type: 'div', key: 'B1', props: { id: 'B1', children: [C1, C2] } };
let B2 = { type: 'div', key: 'B2', props: { id: 'B2', children: [] } };
let A1 = { type: 'div', key: 'A1', 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.key);
        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.setAttribute(key, currentFiber.props[key]);
        }
    }
    let previousFiber;
    currentFiber.props.children.forEach((child, index) => {
        let childFiber = {
            tag: 'HOST',
            type: child.type,
            key: child.key,
            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;
        }
    }
}
console.log(container);

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