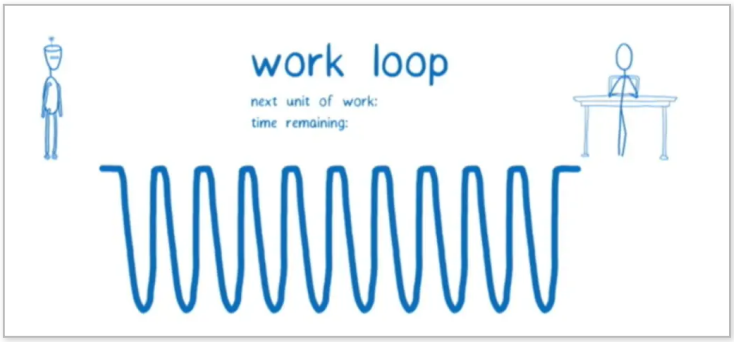
在 Fiber 架构中，React 为什么使用和如何使用单链表

> 本文为意译和整理，如有误导，请放弃阅读。 原文 < https://indepth.dev/the-how-and-why-on-reacts-usage-of-linked-list-in-fiber-to-walk-the-components-tree/>



前言

这篇文章主要是探索 React reconciler 的新实现 – Fiber 中的 work loop。在文本中，我们对比和解释了【浏览器的 call stack】和【React Fiber 架构自己实现的 stack】之间的不同。

正文

为了自学和回馈社区，我花费了大量的时间去做 web 技术的逆向工程 方面的实践，并写文章记录我的发现。在上一年，我主要是聚焦在 Angular 的源码上，在网上发表关于 Angular 方面最大数量的文章 – Angular-In-Depth < https://blog.angularindepth.com/?source=post_page-----------------------> 。现在，是时候要深入到 React 中来了。 Change detection < https://indepth.dev/what-every-front-end-developer-

should-know-about-change-detection-in-angular-and-react/> 是我在 Angular 那边专攻并且有专业水准的领域。希望在足够的耐心和大量的 debugging 下面，我能够早日在 React 这边达到同样的专业水平。

在 React 中，change detection 机制常常被称为“reconciliation”或者“rendering”。而 Fiber 就是 reconciliation 的最新实现。在 Fiber 架构的底层，它为我们提供了实现各种有趣特性的能力。比如说:“非阻塞型的渲染（non-blocking rendering）”，“基于 priority 的更新策略”和“在后台做内容预渲染（pre-rendeing）”。在这些特性在 Concurrent React philosophy < https://twitter.com/acdlite/status/1056612147432574976?source=post_page---------------------------> 中被统称为时间切片 (time slicing)。

除了为应用开发者解决实际的问题外，从（软件）工程学的角度来看，这些机制的内部实现同样是有着强大的吸引力。在源码里面，有着大量的知识能够帮助你成为更加优秀的开发者。

今天，如果你去 google“React Fiber”的话，你将会看到大量的关于这方面的文章。它们之中，除了一篇高质量的 notes by Andrew Clark < https://juejin.im/post/5e7880f45188255e114934b4> ，其他文章的质量嘛..... 你懂的。本文中，我将会引用这篇笔记里面的某些论述。对 Fiber 架构中一些挺特别重要的概念，我会给出更加详尽的解释。一旦你看完并理解了这篇文章，你就能很好地看懂来自于*ReactConf 2017* Lin Clark 的一个很好的演讲 < https://www.youtube.com/watch?v=ZCuYPiUIONs&source=post_page---------------------------> 。 这是一个你需要好好瞧瞧，好好听听的演讲。但是如果你能够花点时间去研究一下源码，然后回来再看这个演讲，效果更佳。

这篇文章了开辟了我的【深入 xxx】系列的先河。我相信，我已经大概弄懂了 Fiber 内部实现细节的 70% 了。于此同时，我准备要写三篇关于 reconciliation 和 rendering 的文章。

下面，让我们开始我们的探索之旅吧。

背景交代

Fiber 架构有两个主要的阶段：

reconciliation/render 阶段和 commit 阶段。在源码中，大部分地方都会把“reconciliation 阶段”称为“render 阶段”。就是在 render 阶段里面，React 遍历了组件树，并且做了以下的这些事情：

- 更新 state 和 props，
- 调用生命周期函数，
- （通过调用 render 方法）从组件中获取 children，
- 把获取到的 children 跟之前的 children 相比
- 最后计算出需要执行的 DOM 操作是什么。

在 Fiber 架构中，所有的这些 activity 都被称为“work”。一个 fiberr node 需要做什么样的 work 取决于其对应的 react element 是什么的类型。打比方说，对于一个 class component 而言，React 需要做的 work 就是实例化这个组件。而对于 functional component 来说，它没有这样的 work 需要去完成。如果你感兴趣的话， 这里 < https://github.com/facebook/react/blob/340bfd9393e8173adca5380e6587e1ea1a23cefa/packages/shared/ReactWorkTags.js?source=post_page----------------------------#L29-L28> 有你想看的所有的 work 的类型。这些 activity 就是 Andrew 在他的笔记中所说的东西：

> When dealing with UIs, the problem is that if too much work is executed all at once, it can cause animations to drop frames...

那上面的“all at once”该如何理解呢？好，基本上，React 会以同步的方式去遍历整一颗组件树，对每个组件进行具体的操作。这样会有什么样的问题呢？实际上，这种方式可能会导

致应用逻辑代码的执行时间超过可用的 16 毫秒。这就会引起界面渲染的掉帧，产生卡顿的视觉效果。

那么，这个问题能被解决吗?

> Newer browsers (and React Native) implement APIs that help address this exact problem...

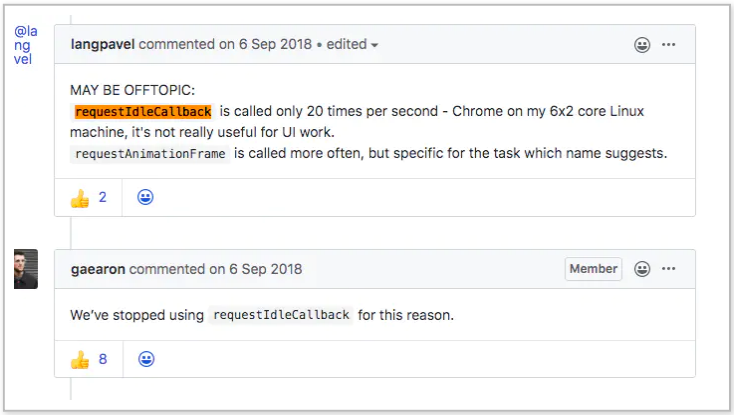
他说的新 API 就是全局函数 requestIdleCallback < https://developers.google.com/web/updates/2015/08/using-requestidlecallback?source=post_page----------------------------------> 。这个全局函数能将一个函数入队起来，然后在浏览器空闲时间里面再去调用它。下面是一个使用的例子：

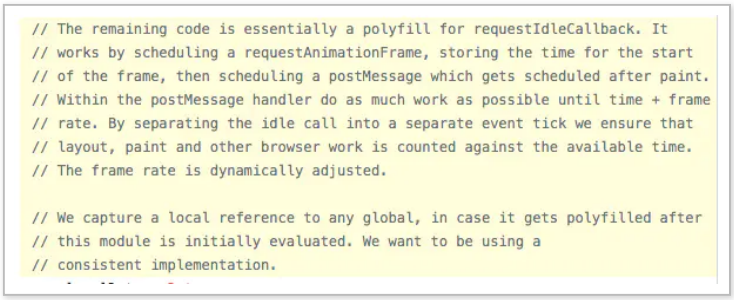
```
requestIdleCallback((deadline)=>{
    console.log(deadline.timeRemaining(), deadline.didTime
});
复制代码
```

如果我在 Chrome 浏览器的控制台上输入以上代码，并执行它。那么，我们会看到控制台打印出 `49.9 false` 。这个运行结果基本上是在告诉我，你有 49.9 毫秒的私人时间去做你自己的事情， `false` 是在告诉我，你没有用完我（指浏览器）分配给你的时间。如果我用完了这个时间，`deadline.didTimeout` 的值将会是 `true` 。需要时刻提醒自己的是，随着浏览器的运行， `timeRemaining` 的字段值会改变的。所以，我们应该经常性地去检查这个字段的值。

> `requestIdleCallback` 实际上的限制比较多和 它的执行频率也不够高 < https://github.com/facebook/react/issues/13206?source=post_page----------------------------------#issuecomment-418923831> ，导致了无法创造一个流畅的界面渲染体验。所以， React 团队不得不实现自己的版本 < https://github.com/facebook/react/blob/eeb817785c771362416fd87ea7d2a1a32dde9842/packages

> /scheduler/src/Scheduler.js?source=post_page---------------------------#L212-L222> 。





现在，假设我们把 React 在组件上需要执行的所有的 work 都放进了 `performWork` 函数里面，然后用 `requestIdleCallback` 去调度这个函数，那么我们的实现代码应该差不多是这样子的：

```
requestIdleCallback((deadline) => {
    // while we have time, perform work for a part of the
    while ((deadline.timeRemaining() > 0 || !deadline.didT
        nextComponent = performWork(nextComponent);
    }
});
```
复制代码

我们一个接一个地在组件上执行 work，然后相继地在处理完当前组件之后把下一个组件的引入返回出去，再接着处理。按理说，这种实现应该是可行的。但是这里有一个问题。你不能像 reconciliation 算法以前的实现那样，采用同步的方式去遍历整一颗组件树。这就是 Andrew 笔记中所提到的问题：

> in order to use those APIs, you need a way to break rendering work into incremental units

因此，为了解决这个问题，React 不得不将遍历组件树所用到的算法重新实现一遍。把它从原来的，依赖于浏览器原生 call stack 的【同步递归模式】改为【用链表和指针实现的异步模式】。Andrew 对于这一点，如是说：

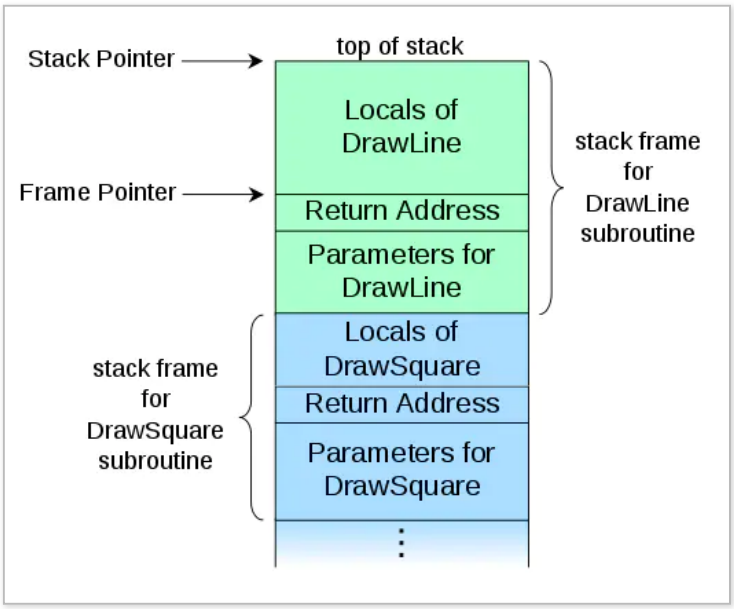
> 如果你仅仅依赖于浏览器原生的 call stack 的话，那么 call stack 会一直执行我们的代码，直到自己被清空为止..... 如果我们能手动地，任意地中断 call stack 和操作 call stack 的每一个帧岂不是很棒吗？而这就是 React Fiber 的目的。React Fiber 是对 stack 的重新实现，特别是为了 React 组件而作的实现。你可以把单独的一个 fiber node 理解为一个 virtual stack frame。

而 Andrew 说的这些话也正是我打算深入解释的东西。

关于 stack 的话

我假设你们都熟悉“call stack”这个概念。它是你在浏览器开发工具打断点的时候调试面板所看到的东西。下面是来自于维基百科的引用和图示：

> 在计算机科学中，call stack 是一种存储计算机程序当前正在执行的子程序（subroutine）信息的栈结构..... 使用 call stack 的主要原因是保存一个用于追踪【当前子程序执行完毕后，程序控制权应该归还给谁】的指针..... 一个 call stack 是由多个 stack frame 组成。每个 stack frame 对应于一个子程序调用。作为 stack frame，这个子程序此时应该还没有被 return 语句所终结。举个例子，我们有一个叫 `DrawLine` 的子程序正在运行。这个子程序又被另外一个叫做 `DrawSquare` 的子程序所调用，那么 call stack 中顶部的布局应该长成下面那样：



为什么 stack 跟 React 有关系?

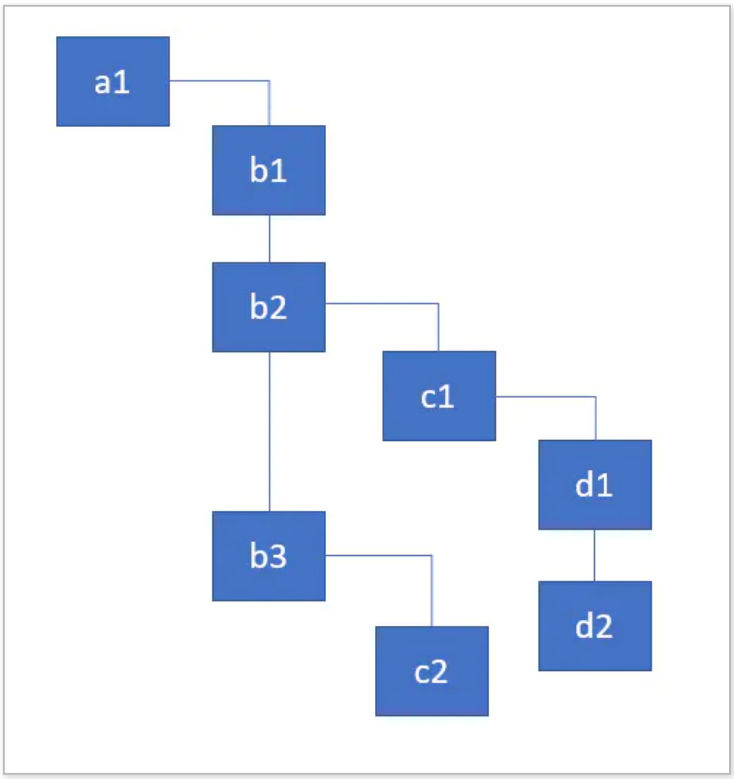
在上面【背景交代】一小节中，我们提到，React 会在 reconciliation/render 阶段遍历整一颗组件树，然后针对每一组件去执行具体的 work。在 reconciler 的先前的实现中，React 使用了【同步递归模式】。这种模式依赖于浏览器原生的 call stack。 这篇官方文档 <https://reactjs.org/docs/reconciliation.html?source=post_page----------#recursing-on-children> 对这个处理流程进行了阐述，并大量地谈到递归：

> By default, when recursing on the children of a DOM node, React just iterates over both lists of children at the same time and generates a mutation whenever there's a difference.

如果你能够对此进行思考的话，你就会知道，每递归调用一次就是往 call stack 上增加一个 stack frame。这样子的话，整个递归流程表现得是如此的同步（太过同步，某种程度下就代

表着阻塞 call stack。想深入 call stack，请查阅我整理的：Event Loop 到底是什么鬼？ <https://juejin.im/post/5e83f9e85188257381151e1d>)。

假设我们有以下的一颗组件树：



我们用一个带有 `render` 方法的 object 去代表每个节点。你也可以把这个 object 当做是组件的实例；

```
const a1 = {name: 'a1'};
const b1 = {name: 'b1'};
const b2 = {name: 'b2'};
const b3 = {name: 'b3'};
const c1 = {name: 'c1'};
const c2 = {name: 'c2'};
const d1 = {name: 'd1'};
const d2 = {name: 'd2'};

a1.render = () => [b1, b2, b3];
b1.render = () => [];
b2.render = () => [c1];
b3.render = () => [c2];
c1.render = () => [d1, d2];
c2.render = () => [];
d1.render = () => [];
d2.render = () => [];
复制代码
```

React 需要迭代整一颗树，对每一个组件执行某些 work。为了简单起见，我们把组件需要执行的 work 定义为“打印组件的名字，并返回 children”。下面一小节就是讲述我们是如何用递归的方式去实现它的。

递归式的遍历

负责对组件树迭代的函数叫做 `walk` 。它的具体实现如下：

```
function walk(instance) {
    doWork(instance);
    const children = instance.render();
    children.forEach(walk);
}

function doWork(o) {
    console.log(o.name);
}

walk(a1);
复制代码
```

执行以上代码，你将会看到以下的输出：

```
a1, b1, b2, c1, d1, d2, b3, c2
复制代码
```

如果你觉得自己对递归的理解不够深入的话，欢迎去阅读我的 深入讲解递归的文章 < https://medium.com/angular-in-depth/learn-recursion-in-10-minutes-e3262ac08a1> 。

在这里使用递归是一种很好的直觉，并且也很适合组件树遍历。但是，我们也发现了它的局限性。其中最大的一点是【我们不能把 work 拆分为增量单元（incremental units）】。我们不能暂停一个特定组件 work 的执行，然后稍后再恢复执行它。递归模式下，React 只能一直迭代下去，直到所有的组件都被处理一遍，call stack 清空了才停下来（此谓之“one pass”）。

那么问题就来了。在不使用递归模式的情况下，React 是如何实现遍历组件树的算法呢？答案是，它使用了单链表（singly-linked-list）式的树遍历算法。这使得遍历暂停和防

止 stack 高度增长成为了可能（stop the stack from growing）。

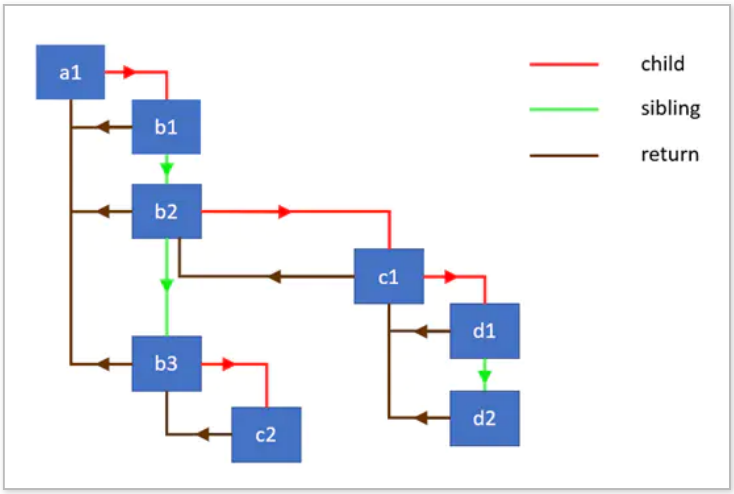
单链表式的遍历

我庆幸自己在 这里 <https://github.com/facebook/react/issues/7942?source=post_page----------------------------#issue-182373497> 找到 Sebastian Markbåge 总结的算法概述。为了实现这个算法，我们需要由三个字段链接起来的数据结构：

- child – 指向第一个 child
- sibling – 指向第一个兄弟节点
- return – 指向 parent（也就是以前 owner 的概念）

在 React 新的 reconciliation 算法这个语境下，具备这三个字段的数据结构被称为“Fiber node”。在底层，它代表着具有 work 需要执行的 react element。想看更多展开的阐述，请看我下一篇文章。

下面这个图演示了 linked-list 中节点的层级和它们之间存在的关系：



下面，让我们一起来定义一下我们自己的 fiber node 的构造函数吧：

```
class Node {
    constructor(instance) {
        this.instance = instance;
        this.child = null;
        this.sibling = null;
        this.return = null;
    }
}
```
复制代码

下面再实现一个将从组件实例的 render 方法返回的 children 链接在一块，使它们成为 linked-list 的函数。这个函数接收一个【parent fiber node】和【由组件实例组成的数组】作为输入，最后返回 parent fiber node 的第一个 child 的引用：

```
function link(parent, elements) {
    if (elements === null) elements = [];

    parent.child = elements.reduceRight((previous, current
        const node = new Node(current);
        node.return = parent;
        node.sibling = previous;
        return node;
    }, null);

    return parent.child;
}
```
复制代码

这个函数从倒数第一个开始（注意看，这里是用了 reduceRight 方法），遍历数组里面的每一个元素，把它们链接成一个 linked-list。最后，把 parent fiber node 的第一个 child fiber node 的引用返回出去。下面这个代码演示一下这个函数的使用：

```
const children = [{name: 'b1'}, {name: 'b2'}];
const parent = new Node({name: 'a1'});
const child = link(parent, children);

// the following two statements are true
console.log(child.instance.name === 'b1');
console.log(child.sibling.instance === children[1]);
```
复制代码

同时，我们也需要实现一个 helper 函数来帮助我们在 fiber node 身上执行具体的 work。在本示例中，这个 work 就是

简单地打印出组件实例的名字。这个 helper 函数除了执行 work 之外，还获取到了组件最新的 children list，然后将他们链接到一块了：

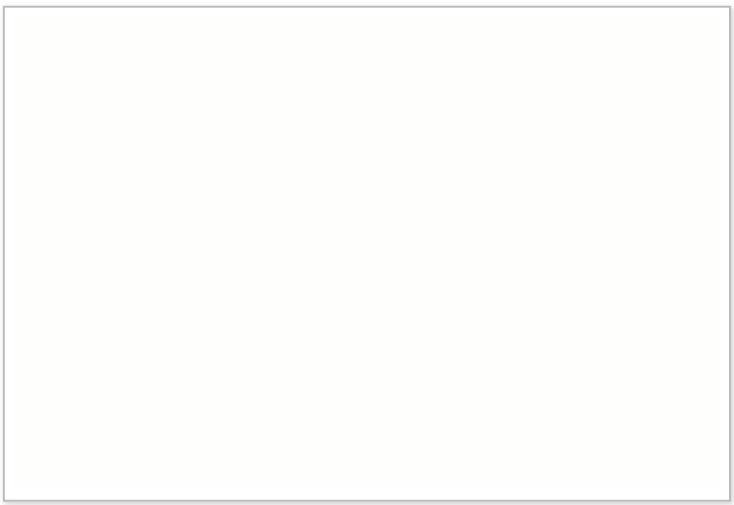
```
function doWork(node) {
    console.log(node.instance.name);
    const children = node.instance.render();
    return link(node, children);
}
复制代码
```

okay，万事俱备只欠东风。下面，我们去实现具体的遍历算法。这是一个深度优先的算法实现。下面是加上了点注释的实现代码：

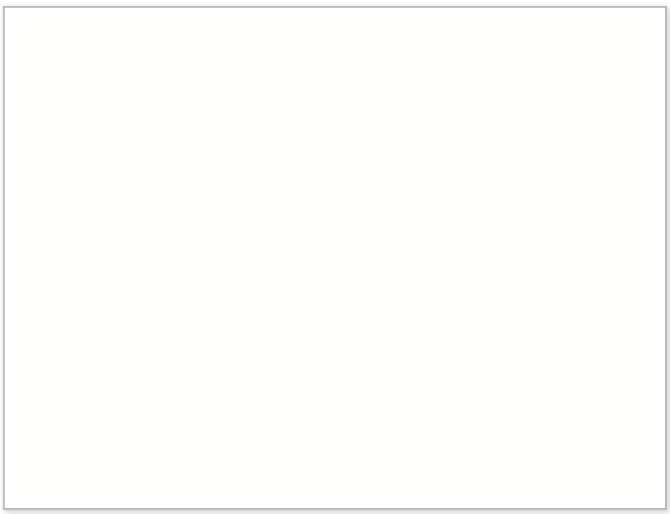
```
 // 参数o你可以说它是一个fiber node也可以说它是一颗fiber node tr
function walk(o) {
    let root = o;
    let current = o;

    while (true) {
        // perform work for a node, retrieve & link the ch
        let child = doWork(current);

        // if there is a child, set it as the current acti
        if (child) {
            current = child;
            continue;
        }

        // if we have returned to the top, exit the functi
        if (current === root) {
            return;
        }

        // keep going up until we find the sibling
        while (!current.sibling) {

            // if we have returned to the top, exit the fu
            if (!current.return || current.return === root
                return;
            }

            // set the parent as the current active node
            current = current.return;
        }

        // if found, set the sibling as the current active
        current = current.sibling;
    }
}
复制代码
```

尽管上面的实现代码不是特别难以理解。但是，我觉得你最好好好把玩一下它，这样你才能理解得更透彻。 Do it here < https://stackblitz.com/edit/js-tle1wr?source=post_page----------------------------> 。这个实现的中心思想是：保留一个指向当前被处理 fiber node 的引用，随着深度优先的向下遍历，不断地修正这个引用，直到遍历触及到这个树分支的叶子节点。一旦到底了，我们就通过 `return` 字段，层层地返回到上一层的 parent fiber node 上去。

如果此时我们去看看这个实现的 call stack 的话，那么我们将会看到这样的画面：



正如你所看到的那样，随着我们的遍历，这个 stack 的高度并没有增加。但是如果我们在 doWork 函数里面打个断点的话，并把组件实例节点的名字打印出来的话，我们将会看到这样的结果：



这个结果的动画跟浏览器 call stack 的表现很像（不同点在于，call stack 的栈底是在下面，而这里是在上面）。有了这个算法实现，我们就能够很好地把浏览器的 call stack 替换为我们自己的 stack。这就是 Andrew 在他的笔记中所讲到的一点：

> Fiber is re-implementation of the stack, specialized for React components. You can think of a single fiber as a virtual stack frame.

现在我们能够保存一个 fiber node 的引用（这个 fiber node 充当着 stack 的 top frame），并通过不断地切换它的指向

> 某种情况下，指向它的 child fiber node，某种情况下指向它的 sibling fiber node，某种情况下指向它的 return/parent fiber node

来控制我们的“call stack”了：

```
function walk(o) {
    let root = o;
    let current = o;

    while (true) {
            ...

            current = child;
            ...

            current = current.return;
            ...

            current = current.sibling;
    }
}
```
复制代码

因此，我们能够在遍历过程中随意地暂停和恢复执行。而这也是能够使用新的 `requestIdleCallback` API 的先决条件。

React 中的 work loop

下面是 React 中实现 work loop 的代码：

```
function workLoop(isYieldy) {
    if (!isYieldy) {
        // Flush work without yielding
        while (nextUnitOfWork !== null) {
            nextUnitOfWork = performUnitOfWork(nextUnitOfW
        }
    } else {
        // Flush asynchronous work until the deadline runs
        while (nextUnitOfWork !== null && !shouldYield())
            nextUnitOfWork = performUnitOfWork(nextUnitOfW
        }
    }
}
```
复制代码

正如你所看到的那样，React 中的实现跟我们上面提到的算法实现十分相似。它也是通过 `nextUnitOfWork` 变量来保存一个【代表 top frame 的】fiber node 引用。

React 实现的 walk loop 算法能以同步的方式去遍历组件树，并在每个 fiber node 上（nextUnitOfWork）执行某些 work。同步的方式往往是发生在所谓的【由于 UI 事件（比如，click，input 等）的发生而导致的】“交互式更新”场景下。除了同步方式，walk loop 也能够以异步的方式去进行。在遍历过程中，在每执行一个 fiber node 相关 work 之前，该算法会去检查当前是否还有可用时间。 `shouldYield` 函数会基于 deadlineDidExpire < https://github.com/facebook/react/blob/95a313ec0b957f71798a69d8e83408f40e76765b/packages/react-reconciler/src/ReactFiberScheduler.js?source=post_page---------------------------#L1806> 和 deadline < https://github.com/facebook/react/blob/95a313ec0b957f71798a69d8e83408f40e76765b/packages/react-reconciler/src/ReactFiberScheduler.js?source=post_page---------------------------#L1809> 的变量值去返回结果。这两个变量的值会随着 React 对 fiber node 的 work 执行的推进而随时被更新的。

想要了解 `peformUnitOfWork` 的更多细节，请查阅这篇文章： 深入 React Fiber 架构的 reconciliation 算法 < https://juejin.im/post/5e92e592f265da48027a2c5d> 。

全文完