这可能是最通俗的 React Fiber(时间分片) 打开方式

写一篇关于 React Fiber 的文章， 这个 Flag 立了很久，这也是今年的目标之一。 最近的在掘金的文章获得很多关注和鼓励，给了我很多动力，所以下定决心好好把它写出来。 我会以最通俗的方式将它讲透, 因此这算是一篇科普式的文章。不管你是使用React、还是Vue，这里面的思想值得学习学习!

一年一度的 React 春晚: React Conf 即将到来，不知道今年会不会有什么惊喜，去年是 React Hooks，前年是 React Fiber... 我得赶在 React Conf 之前发布这篇文章:

- 😲 **React Fiber 已经出来这么久了， 这文章是老酒装新瓶吧?** *对于我来说，通过这篇文章我重新认识了 React Fiber，它不是一个新东西, 它也是老酒装新瓶，不信你就看吧...*
- 🆕 **React Fiber 不是一个新的东西，但在前端领域是第一次广为认知的应用。**
- 😦 **了解它有啥用?** *React Fiber 代码很复杂，门槛很高，你不了解它，后面 React 新出的 Killer Feature 你可能就更不能理解了*
- 😧 **我不是升到React v16了吗? 没什么出奇的啊?** *真正要体会到 React Fiber 重构效果，可能下个月、可能要等到 v17。v16 只是一个过渡版本，也就是说，现在的React 还是同步渲染的，一*
  跳票、不是说今年第二季度就出来了吗?
- 😁 **不好意思，一不小心又写得有点长，你就当小说看吧, 代码都是伪代码**

以下文章大纲

单处理进程调度: Fiber 不是一个新的东西

这个黑乎乎的界面应该就是微软的 `DOS` 操作系统

微软 `DOS` 是一个 单任务操作系统，也称为'单工操作系统'. 这种操作系统同一个时间只允许运行一个程序. invalid s在《在没有GUI的时代(只有一个文本界面），人们是怎么运行多个程序的？》 的回答中将其称为: '**一种压根没有任务调度的“残疾”操作系统**'.

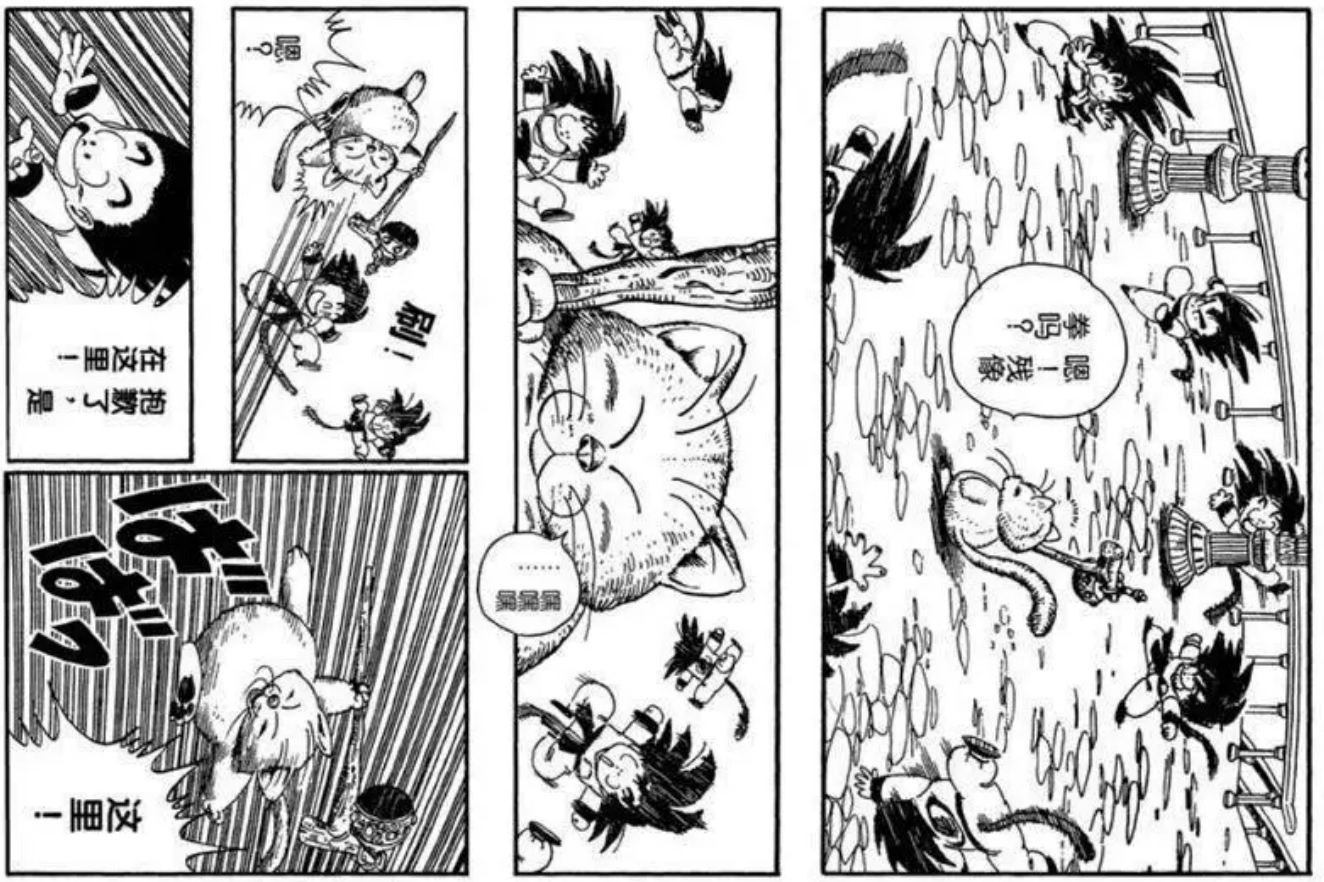
在这种系统中，你想执行多个任务，只能等待前一个进程退出，然后再载入一个新的进程。

直到 Windows 3.x，它才有了真正意义的进程调度器，实现了多进程并发执行。

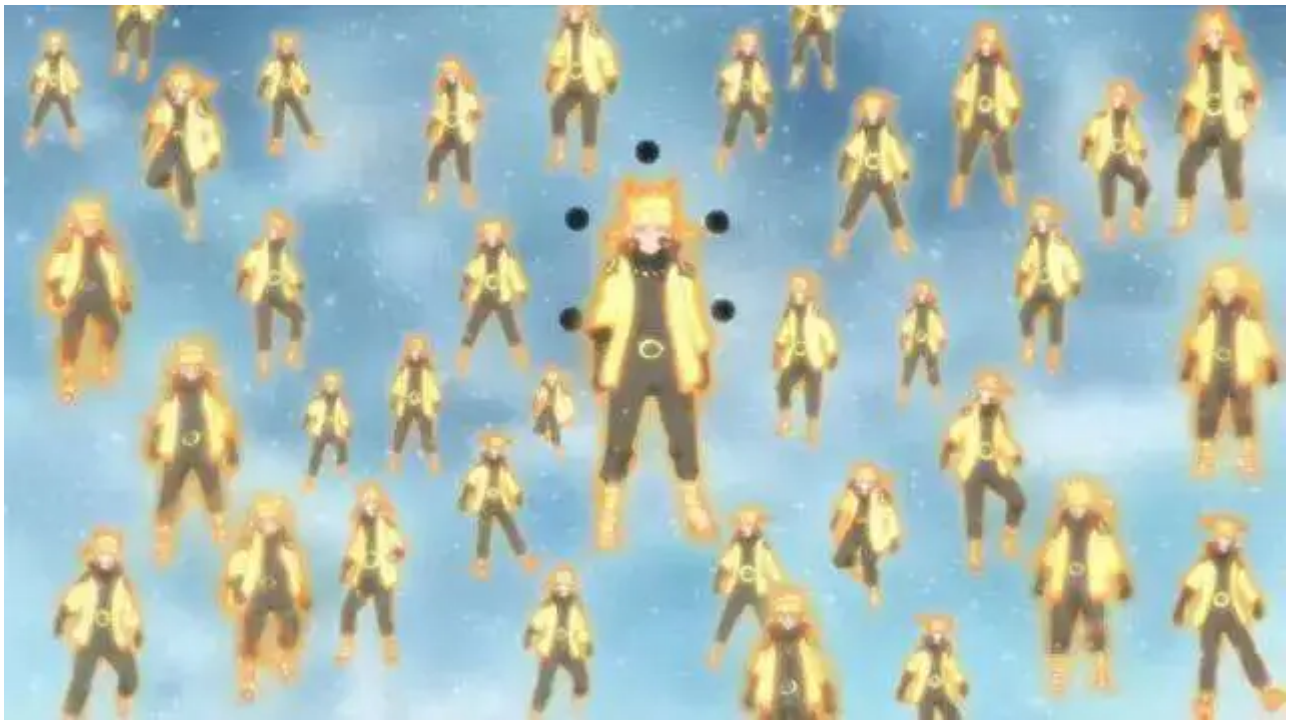
> 注意并发和并行不是同一个概念。

现代操作系统都是**多任务操作系统**. 进程的调度策略如果按照CPU核心数来划分，可以分为**单处理器调度**和**多处理器调度**。本文只关注的是单处理器调度，因为它可以类比JavaScript的运行机制。

🔴 **说白了，为了实现进程的并发，操作系统会按照一定的调度策略，将CPU的执行权分配给多个进程，多个进程都有被执行的机会，让它们交替执行，形成一种“同时在运行”假象, 因为CPU速度太快，人类根本感觉不到。实际上在单核的物理环境下同时只能有一个程序在运行。**



这让我想起了“龙珠”中的分身术(小时候看过，说错了别喷)，实质上是一个人，只不过是他运动速度太快，看起来就像分身了. 这就是所谓的**并发(Concurrent)**(单处理器)。



相比而言, 火影忍者中的分身术，是物理存在的，他们可以真正实现同时处理多个任务，这就是**并行**(严格地讲这是 `Master-Slave` 架构，分身虽然物理存在，但应该没有独立的意志)。

所以说🔴 **并行可以是并发，而并发不一定是并行，两种不能划等号, 并行一般需要物理层面的支持。**关于并发和并行，Go 之父 Rob Pike 有一个非常著名的演讲Concurrency is not parallelism

扯远了，接下来进程怎么调度就是教科书的内容了。如果读者在大学认真学过**操作系统原理**, 你可以很快理解以下几种单处理器进程**调度策略**(我就随便科普一下，算送的, 如果你很熟悉这块，可以跳过)：

0️⃣ 先到先得(First-Come-First-Served, FCFS)

这是最简单的调度策略, 简单说就是**没有调度**。谁先来谁就先执行，执行完毕后就执行下一个。不过如果中间某些进程因为I/O阻塞了，这些进程会挂起移回就绪队列(说白了就是重新排队).

`FCFS` 上面 `DOS` 的单任务操作系统没有太大的区别。所以非常好理解，因为生活中到处是这样的例子:。

- **FCFS 对 短进程 不利。** 短进程即执行时间非常短的进程，可以用饭堂排队来比喻: *在饭堂排队的时候，最烦那些一个人打包好好几份的人，这些人就像 长进程 一样，霸占着CPU资源，后面*

队只打一份的人会觉得很吃亏，打一份的人会觉得他们优先级应该更高，毕竟他们花的时间很短，反正你打包那么多份再等一会也是可以的，何必让后面那么多人等这么久...

- **FCFS 对 `I/O密集` 不利**。I/O密集型进程(这里特指同步I/O)在进行I/O操作时，会阻塞休眠，这会导致进程重新被放入就绪队列，等待下一次被宠幸。 可以类比ZF部门办业务: *假设 CPU 一个窗口、I/O 一个窗口。在CPU窗口好不容易排到你了，这时候发现一个不符合条件或者漏办了, 需要去I/O搞一下，Ok 去 I/O窗口排队，I/O执行完了，到CPU窗口又得重新排队。对于这些丢三落四的人很不公平...*

所以 FCFS 这种原始的策略在单处理器进程调度中并不受欢迎。

1️⃣ 轮转

这是一种基于时钟的**抢占策略**，这也是抢占策略中最简单的一种: **公平地给每一个进程一定的执行时间，当时间消耗完毕或阻塞，操作系统就会调度其他进程，将执行权抢占过来。**

> **决策模式**: `抢占策略` 相对应的有 `非抢占策略` ，非抢占策略指的是让进程运行直到结束、阻塞(如I/O或睡眠)、或者主动让出控制权；抢占策略支持中断正在运行的进程，将主动权掌握在操作系统这里，不过通常开销会比较大。

这种调度策略的要点是**确定合适的时间片长度**: 太长了，长进程霸占太久资源，其他进程会得不到响应(等待执行时间过长)，这时候就跟上述的 `FCFS` 没什么区别了; 太短了也不好，因为进程抢占和切换都是需要成本的, 而且成本不低，时间片太短，时间可能都浪费在上下文切换上了，导致进程干不了什么实事。

因此**时间片的长度最好符合大部分进程完成一次典型交互所需的时间**.

轮转策略非常容易理解，只不过确定时间片长度需要伤点脑筋；另外和 `FCFS` 一样，轮转策略对I/O进程还是不公平。

2️⃣ 最短进程优先(Shortest Process Next, SPN)

上面说了 `先到先得` 策略对 `短进程` 不公平， `最短进程优先` 索性就让'最短'的进程优先执行，也就是说: **按照进程的预估执行时间对进程进行优先级排序，先执行完短进程，后执行长进程。这是一种非抢略。**

这样可以让短进程能得到较快的响应。但是怎么获取或者**评估进程执行时间**呢？一是让程序的提供者提供，这不太靠谱；二是由操作系统来收集进程运行数据，并对它们进程统计分析。例如最简单的是计算它们的平均运行时间。不管怎么说都比上面两种策略要复杂一点。

`SPN` 的缺陷是: 如果系统有大量的短进程，那么长进程可能会饥饿得不到响应。

另外因为它不是抢占性策略, 尽管现在短进程可以得到更多的执行机会，但是还是没有解决 `FCFS` 的问题: 一旦长进程得到CPU资源，得等它执行完，导致后面的进程得不到响应。

3️⃣ 最短剩余时间(Shortest Remaining Time, SRT)

SRT 进一步优化了SPN，增加了抢占机制。在 SPN 的基础上，当一个进程添加到就绪队列时，操作系统会比较*刚添加的新进程*和*当前正在执行的老进程*的'剩余时间'，如果新进程剩余时间更短，新进程就会抢占老进程。

相比轮转的抢占，SRT 没有中断处理的开销。但是在 SPN 的基础上，操作系统需要记录进程的历史执行时间，这是新增的开销。**另外长进程饥饿问题还是没有解决**。

4️⃣ 最高响应比优先(HRRN)

为了解决长进程饥饿问题，同时提高进程的响应速率。还有一种 `最高响应比优先的` 策略，首先了解什么是响应比:

```shell
响应比 = （等待执行时间 + 进程执行时间） / 进程执行时间
```

这种策略会选择响应比最高的进程优先执行:

- 对于短进程来说，因为执行时间很短，分母很小，所以响应比比较高，会被优先执行
- 对于长进程来说，执行时间长，一开始响应比小，但是随着等待时间增长，它的优先级会越来越高，最终可以被执行

5️⃣ 反馈法

SPN、SRT、HRRN都需要对进程时间进行评估和统计，实现比较复杂且需要一定开销。而反馈法采取的是**事后反馈**的方式。这种策略下: **每个进程一开始都有相同的优先级，每次被抢占(需要配合其他抢占策略使用，如轮转)，优先级就会降低一级。因此通常它会根据优先级划分多个队列**。

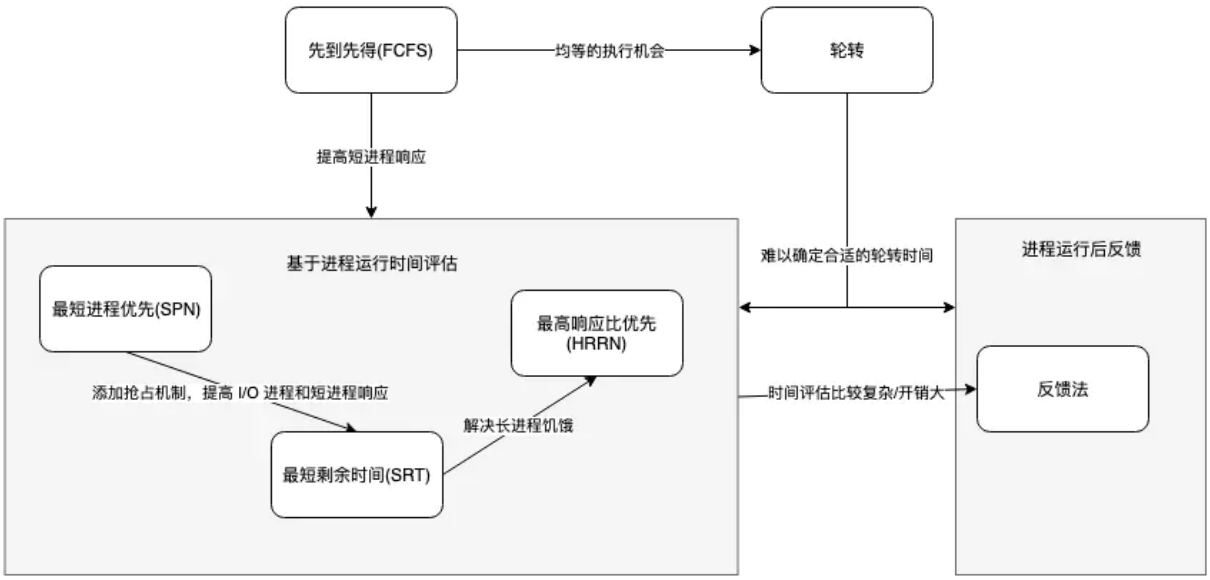
举个例子:

```shell
队列1
队列2
...
队列N
```

新增的任务会推入 队列1 ， 队列1 会按照 轮转策略 以一个时间片为单位进行调度。短进程可以很快得到响应，而对于长进程可能一个时间片处理不完，就会被抢占，放入 队列2 。

队列2 会在 队列1 任务清空后被执行，有时候低优先级队列可能会等待很久才被执行，所以一般会给予一定的补偿，例如增加执行时间，所以 队列2 的轮转时间片长度是2。

反馈法仍然可能导致长进程饥饿，所以操作系统可以统计长进程的等待时间，当等待时间超过一定的阈值，可以选择提高它们的优先级。



没有一种调度策略是万能的, 它需要考虑很多因素:

- 响应速率。进程等待被执行的时间
- 公平性。兼顾短进程、长进程、I/O进程

这两者在某些情况下是对立的，提高了响应，可能会减低公平性，导致饥饿。短进程、长进程、I/O进程之间要取得平衡也非常难。

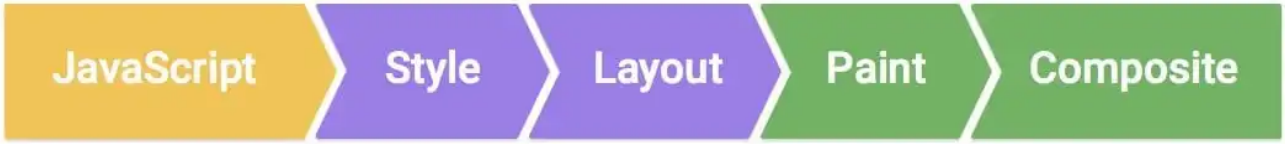
上面这些知识对本文来说已经足够了，现实世界操作系统的进程调度算法比教科书上说的要复杂的多，有兴趣读者可以去研究一下 `Linux` 相关的进程调度算法，这方面的资料也非常多, 例如《Linux进程调度策略的发展和演变》。

类比浏览器JavaScript执行环境

JavaScript 就像单行道

JavaScript 是单线程运行的，而且在浏览器环境屁事非常多，它要负责页面的JS解析和执行、绘制、事件处理、静态资源加载和处理, 这些任务可以类比上面'进程'。

> 这里特指Javascript 引擎是单线程运行的。 严格来说，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. 另外异步 I/O 操作底层实际上可能是多线程的在驱动。



图片来源: Rendering Performance

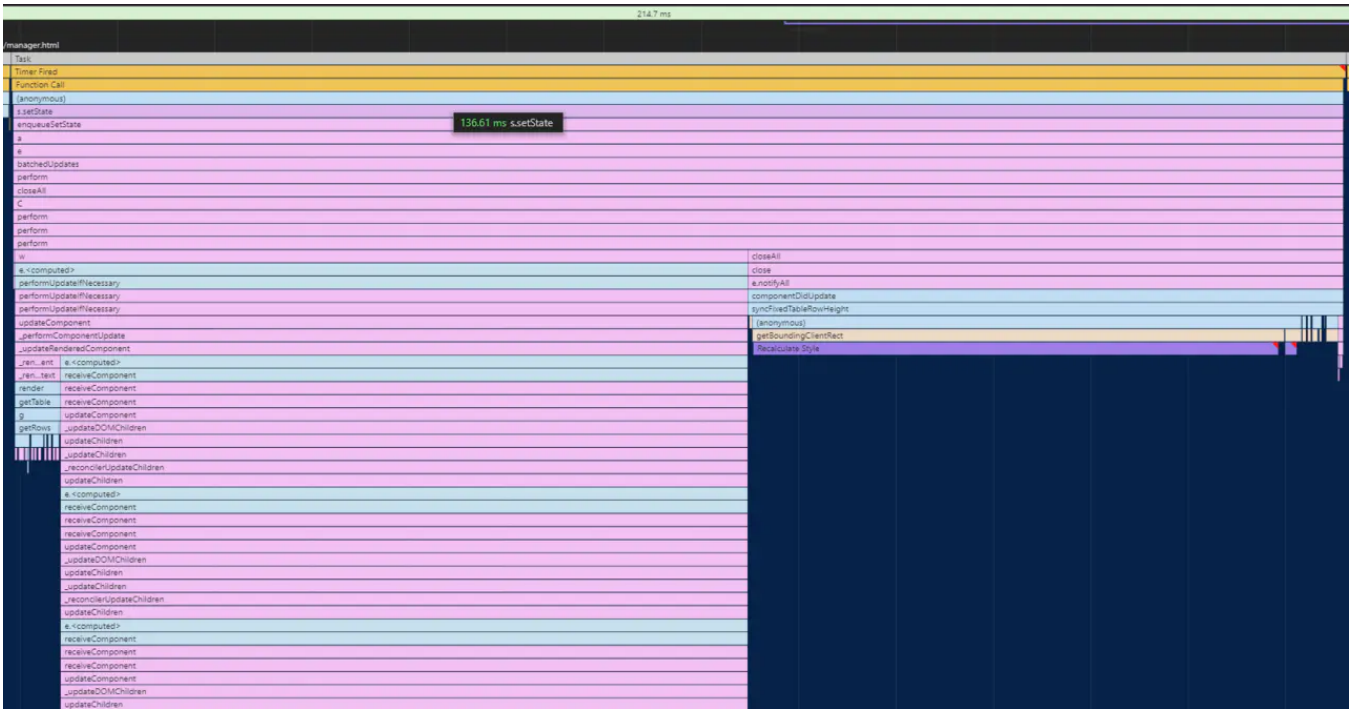
它只是一个'JavaScript'，同时只能做一件事情，这个和 `DOS` 的单任务操作系统一样的，事情只能一件一件的干。要是前面有一个傻叉任务长期霸占CPU，后面什么事情都干不了，浏览器会呈现卡死的状态，这样的用户体验就会非常差。

对于'前端框架'来说，解决这种问题有三个方向:

- 1️⃣ 优化每个任务，让它有多快就多快。挤压CPU运算量
- 2️⃣ 快速响应用户，让用户觉得够快，不能阻塞用户的交互
- 3️⃣ 尝试 Worker 多线程

Vue 选择的是第1️⃣ , 因为对于Vue来说，使用 模板 让它有了很多优化的空间，配合响应式机制可以让Vue可以精确地进行节点更新, 读者可以去看一下今年Vue Conf 尤雨溪的演讲，非常棒!；而 React 选择了2️⃣ 。对于Worker 多线程渲染方案也有人尝试，要保证状态和视图的一致性相当麻烦。

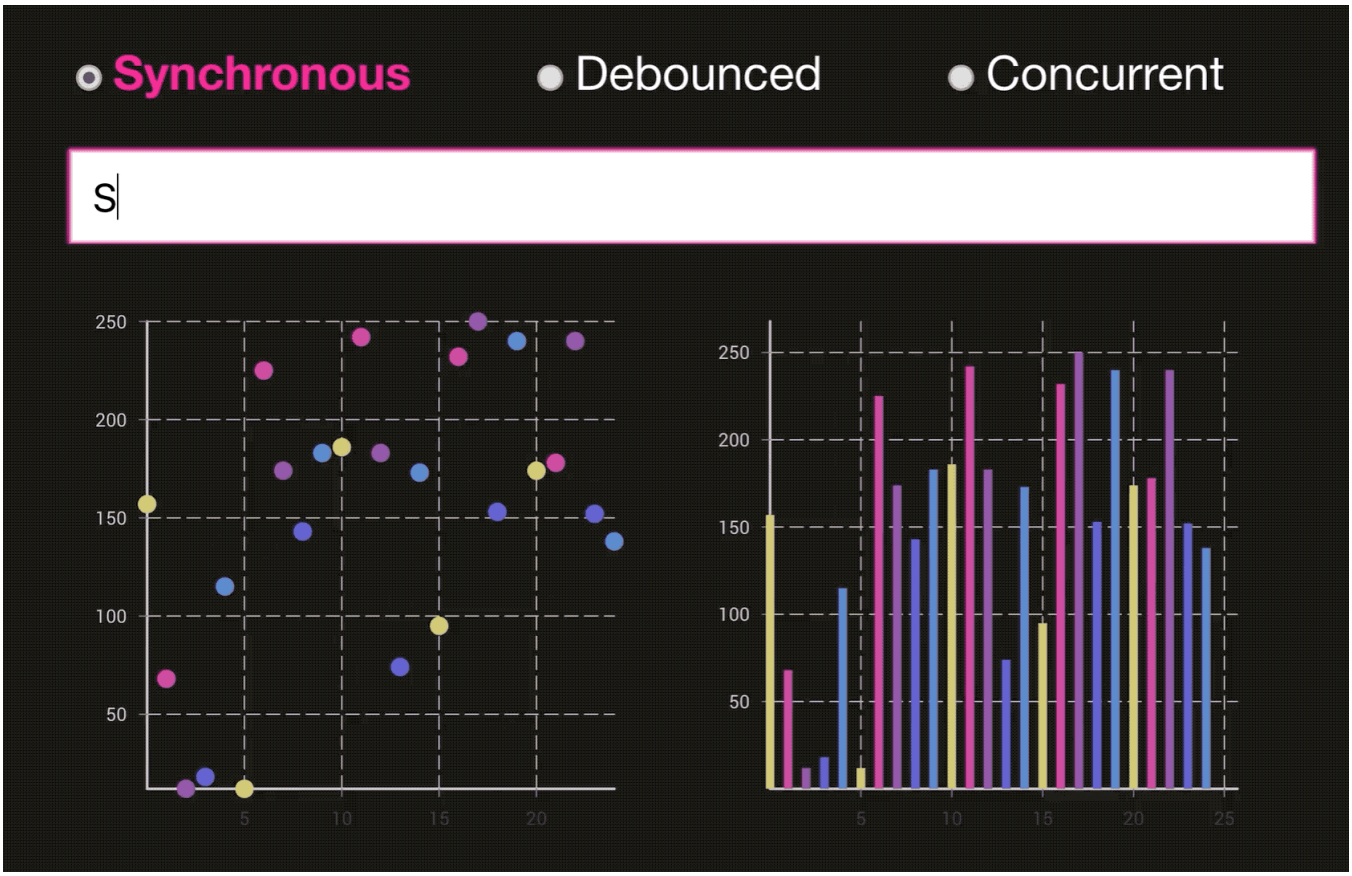
React 为什么要引入 Fiber 架构？ 看看下面的火焰图，这是React V15 下面的一个列表渲染资源消耗情况。整个渲染花费了130ms, 🔴 **在这里面 React 会递归比对VirtualDOM树，找出需要变动的节点，然后同步更新它们, 一气呵成。这个过程 React 称为 `Reconcilation` (中文可以译为 协调).**



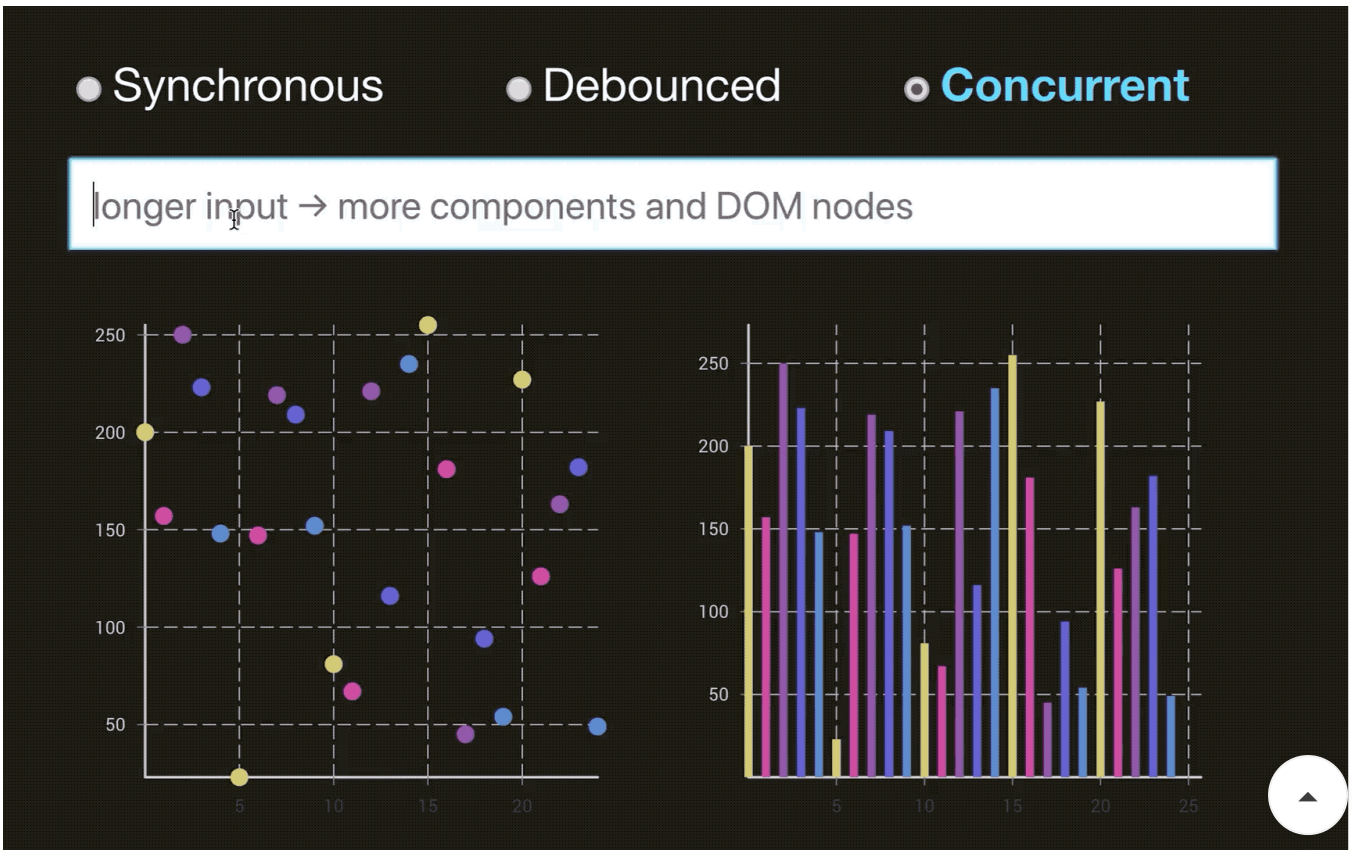
在 Reconcilation 期间，React 会霸占着浏览器资源，一则会导致用户触发的事件得不到响应, 二则会导致掉帧，用户可以感知到这些卡顿。

这样说，你可能没办法体会到，通过下面两个图片来体会一下(*图片来源于：Dan Abramov 的 Beyond React 16 演讲, 推荐看一下👍. 另外非常感谢淡苍 将一个类似的DEMO 分享在了 CodeSandbox上🎉，大家自行体验*):

同步模式下的 React:



优化后的 `Concurrent` 模式下的 React:



React 的 Reconcilation 是CPU密集型的操作, 它就相当于我们上面说的'长进程'。所以初衷和进程调度一样，我们要让高优先级的进程或者短进程优先运行，不能让长进程长期霸占资源。

所以React 是怎么优化的？ 划重点， 🔴 **为了给用户制造一种应用很快的'假象'，我们不能让一个程序长期霸占着资源. 你可以将浏览器的渲染、布局、绘制、资源加载(例如HTML解析)、事件响应、脚本执行视作操作系统的'进程'，我们需要通过某些调度策略合理地分配CPU资源，从而提高浏览器的用户响应速率, 同时兼顾任务执行效率。**

🔴 **所以 React 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，还有其他好处:**

- 与其一次性操作大量 DOM 节点相比, 分批延时对DOM进行操作，可以得到更好的用户体验。这个在《「前端进阶」高性能渲染十万条数据(时间分片)》 以及司徒正美的《React Fiber架构》 都做了相关实验
- 司徒正美在《React Fiber架构》 也提到：🔴 **给浏览器一点喘息的机会，他会对代码进行编译优化（JIT）及进行热代码优化，或者对reflow进行修正.**

这就是为什么React 需要 Fiber 😏 。

何为 Fiber

对于 React 来说，Fiber 可以从两个角度理解:

1. 一种流程控制原语

Fiber 也称协程、或者纤程。 笔者第一次接触这个概念是在学习 Ruby 的时候，Ruby就将协程称为 Fiber。后来发现很多语言都有类似的机制，例如Lua 的 `Coroutine` , 还有前端开发者比较熟悉的 `ES6` 新增的 `Generator` 。

> 本文不纠结 Processes, threads, green threads, protothreads, fibers, coroutines: what's th difference?

🔴 **其实协程和线程并不一样，协程本身是没有并发或者并行能力的（需要配合线程），它只是一种控制流程的让出机制**。要理解协程，你得和普通函数一起来看, 以Generator为例:

普通函数执行的过程中无法**被中断和恢复**:

```js
const tasks = []
function run() {
  let task
  while (task = tasks.shift()) {
    execute(task)
  }
}
```

而 `Generator` 可以:

```js
const tasks = []
function * run() {
  let task

  while (task = tasks.shift()) {
    // 🔴 判断是否有高优先级事件需要处理，有的话让出控制权
    if (hasHighPriorityEvent()) {
      yield
    }

    // 处理完高优先级事件后，恢复函数调用栈，继续执行...
    execute(task)
  }
}
```

React Fiber 的思想和协程的概念是契合的: 🔴 **React 渲染的过程可以被中断，可以将控制权交回浏览器，让位给高优先级的任务，浏览器空闲后再恢复渲染**。

那么现在你应该有以下疑问:

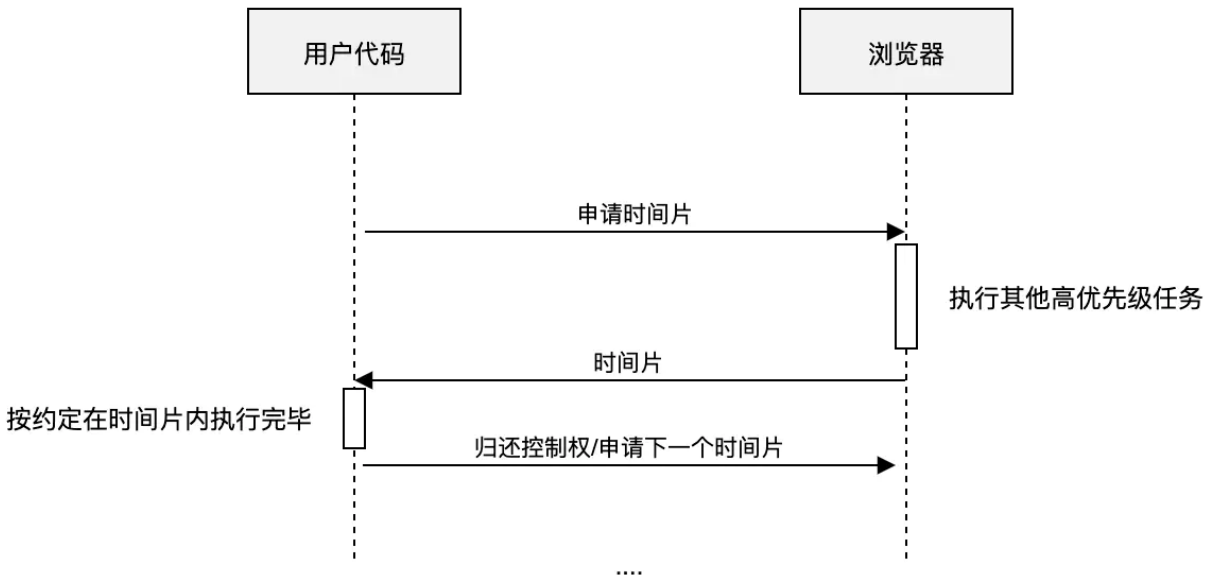
- 1️⃣ 浏览器没有抢占的条件, 所以React只能用让出机制?
- 2️⃣ 怎么确定有高优先任务要处理，即什么时候让出?
- 3️⃣ React 那为什么不使用 Generator?

答1️⃣ : 没错, 主动让出机制

一是浏览器中没有类似进程的概念，'任务'之间的界限很模糊，没有上下文，所以不具备中断/恢复的条件。二是没有抢占的机制，我们无法中断一个正在执行的程序。

所以我们只能采用类似协程这样控制权让出机制。这个和上文提到的进程调度策略都不同，它有更一个专业的名词：**合作式调度(Cooperative Scheduling)**, 相对应的有**抢占式调度(Preemptive Scheduling)**

这是一种'契约'调度，要求我们的程序和浏览器紧密结合，互相信任。比如可以由浏览器给我们分配执行时间片(通过 `requestIdleCallback` 实现, 下文会介绍)，我们要按照约定在这个时间内执行完毕，并将控制权还给浏览器。



这种调度方式很有趣，你会发现**这是一种身份的对调**，以前我们是老子，想怎么执行就怎么执行，执行多久就执行多久; 现在为了我们共同的用户体验统一了战线, 一切听由浏览器指挥调度，浏览器是老子，我们要跟浏览器申请执行权，而且这个执行权有期限，借了后要按照约定归还给浏览器。

当然你超时不还浏览器也拿你没办法 🤷‍♀️ ... 合作式调度的缺点就在于此，全凭自律，用户要挖大坑都拦不住。

答2️⃣ : requestIdleCallback API

上面代码示例中的 `hasHighPriorityEvent()` 在目前浏览器中是无法实现的，我们没办法判断当前是否有更高优先级的任务等待被执行。

只能换一种思路，通过**超时检查的机制来让出控制权**。解决办法是: *确定一个合理的运行时长，然后在合适的检查点检测是否超时(比如每执行一个小任务)，如果超时就停止执行，将控制权交换给浏览器。*

举个例子，为了让视图流畅地运行，可以按照人类能感知到最低限度每秒60帧的频率划分时间片，这样每个时间片就是 16ms。

其实浏览器提供了相关的接口 —— `requestIdleCallback` API:

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

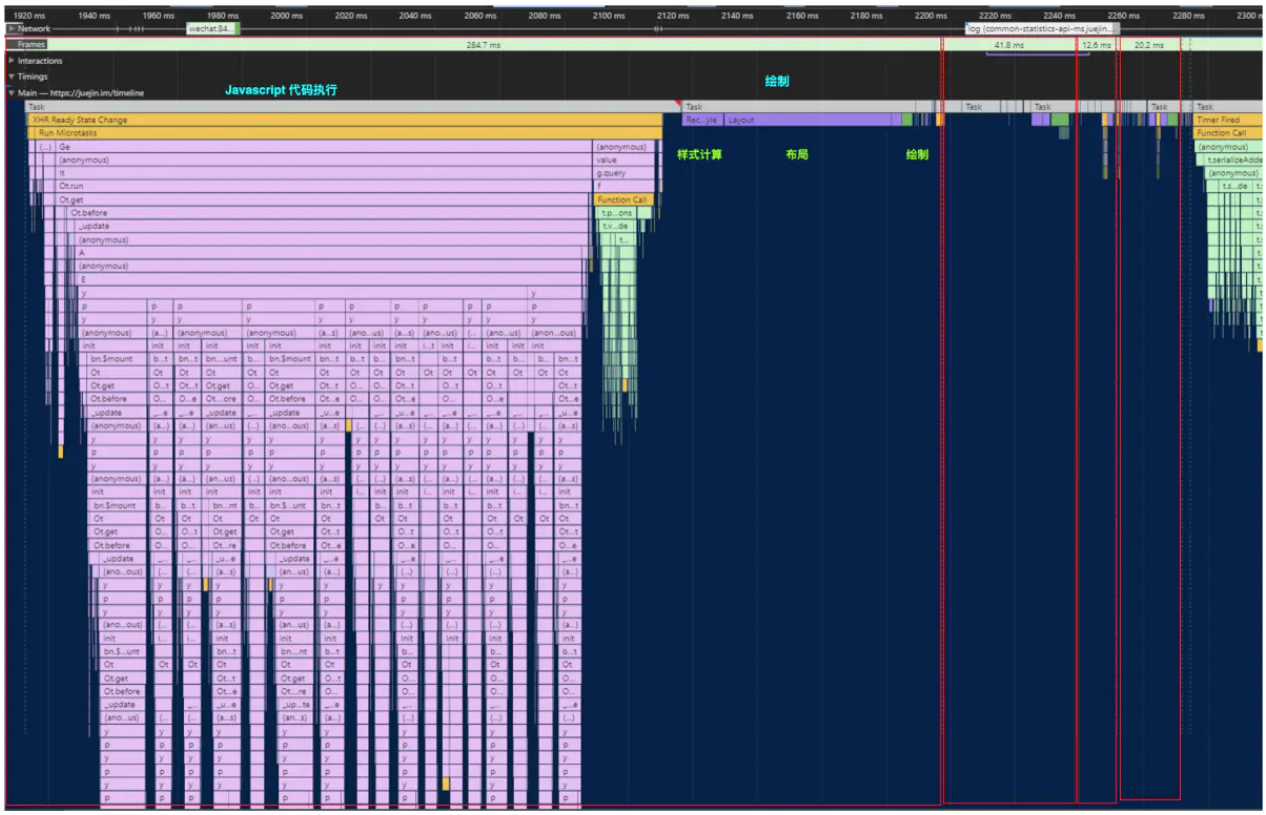
`IdleDeadline` 的接口如下:

```ts
interface IdleDealine {
  didTimeout: boolean // 表示任务执行是否超过约定时间
  timeRemaining(): DOMHighResTimeStamp // 任务可供执行的剩余时间
}
```

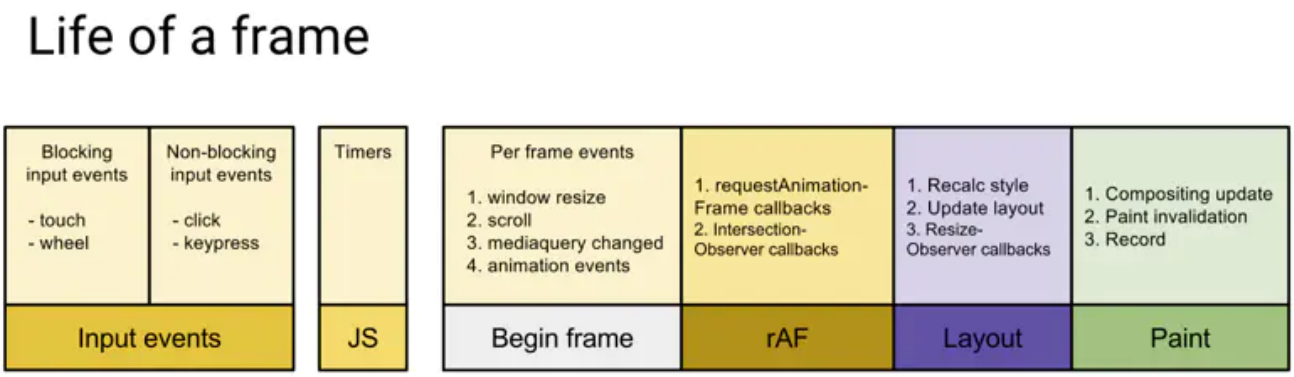
单从名字上理解的话，`requestIdleCallback` 的意思是**让浏览器在'有空'的时候就执行我们的回调，这个回调会传入一个期限，表示浏览器有多少时间供我们执行, 为了不耽误事，我们最好在这个时间范围内执行完毕**。

那浏览器什么时候有空?

我们先来看一下浏览器在一帧(Frame，可以认为事件循环的一次循环)内可能会做什么事情:



你可以打开 Chrome 开发者工具的Performance标签，这里可以详细看到Javascript的每一帧都执行了什么任务(Task)，花费了多少时间。



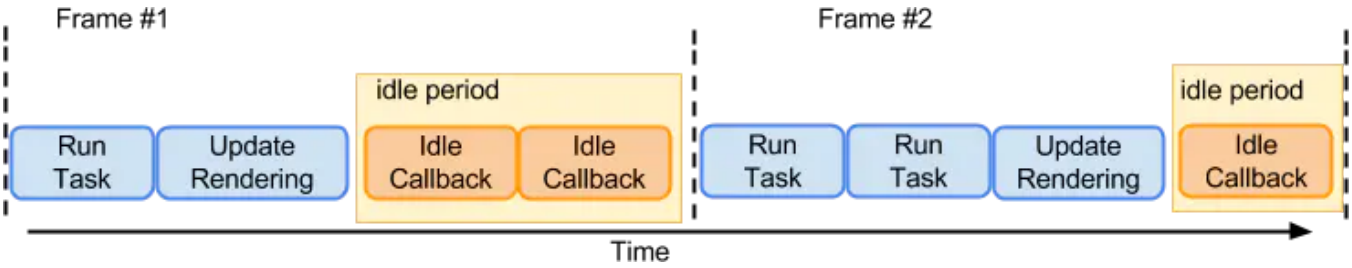
图片来源: 你应该知道的requestIdleCallback

浏览器在一帧内可能会做执行下列任务，而且它们的执行顺序基本是固定的:

- 处理用户输入事件
- Javascript执行
- requestAnimation 调用
- 布局 Layout

- 绘制 Paint

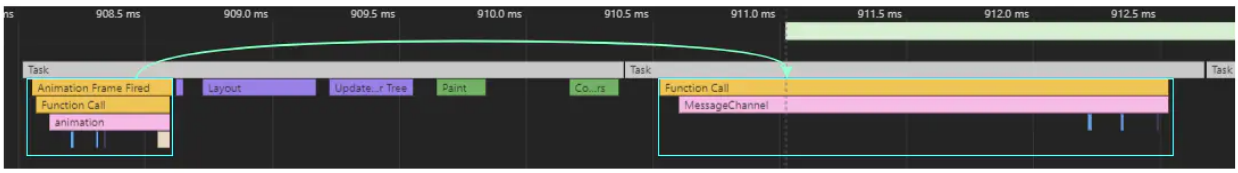
上面说理想的一帧时间是 `16ms` (1000ms / 60)，如果浏览器处理完上述的任务(布局和绘制之后)，还有盈余时间，浏览器就会调用 `requestIdleCallback` 的回调。例如



但是在浏览器繁忙的时候，可能不会有盈余时间，这时候 `requestIdleCallback` 回调可能就不会被执行。 为了避免饿死，可以通过requestIdleCallback的第二个参数指定一个超时时间。

> 另外不建议在 `requestIdleCallback` 中进行 `DOM` 操作，因为这可能导致样式重新计算或重新布局(比如操作DOM后马上调用 `getBoundingClientRect`)，这些时间很难预估的，很有可能导致回调执行超时，从而掉帧。

目前 `requestIdleCallback` 目前只有Chrome支持。所以目前 React 自己实现了一个。它利用 `MessageChannel` 模拟将回调延迟到'绘制操作'之后执行:



▶ 简单看一下代码

任务优先级

上面说了，为了避免任务被饿死，可以设置一个超时时间. **这个超时时间不是死的，低优先级的可以慢慢等待, 高优先级的任务应该率先被执行**. 目前 React 预定义了 5 个优先级, 这个我在[《谈谈React事件机制和未来(react-events)》]中也介绍过:

- `Immediate` (-1) - 这个优先级的任务会同步执行, 或者说要马上执行且不能中断
- `UserBlocking` (250ms) 这些任务一般是用户交互的结果, 需要即时得到反馈
- `Normal` (5s) 应对哪些不需要立即感受到的任务，例如网络请求
- `Low` (10s) 这些任务可以放后，但是最终应该得到执行. 例如分析通知
- `Idle` (没有超时时间) 一些没有必要做的任务 (e.g. 比如隐藏的内容), 可能会被饿死

答3️⃣：太麻烦

官方在《Fiber Principles: Contributing To Fiber》也作出了解答。主要有两个原因：

1. Generator 不能在栈中间让出。比如你想在嵌套的函数调用中间让出, 首先你需要将这些函数都包装成Generator，另外这种栈中间的让出处理起来也比较麻烦，难以理解。除了语法开销，现有的生成器实现开销比较大，所以不如不用。
2. Generator 是有状态的, 很难在中间恢复这些状态。

> 上面理解可能有出入，建议看一下原文

可能都没看懂，简单就是 React 尝试过用 Generator 实现，后来发现很麻烦，就放弃了。

2. 一个执行单元

Fiber的另外一种解读是'纤维': **这是一种数据结构或者说执行单元**。我们暂且不管这个数据结构长什么样，🔴 **将它视作一个执行单元，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**.

上文说了，React 没有使用 Generator 这些语言/语法层面的让出机制，而是实现了自己的调度让出机制。这个机制就是基于'Fiber'这个执行单元的，它的过程如下：

假设用户调用 `setState` 更新组件, 这个待更新的任务会先放入队列中, 然后通过 `requestIdleCallback` 请求浏览器调度：

```js
updateQueue.push(updateTask);
requestIdleCallback(performWork, {timeout});
```

现在浏览器有空闲或者超时了就会调用 `performWork` 来执行任务：

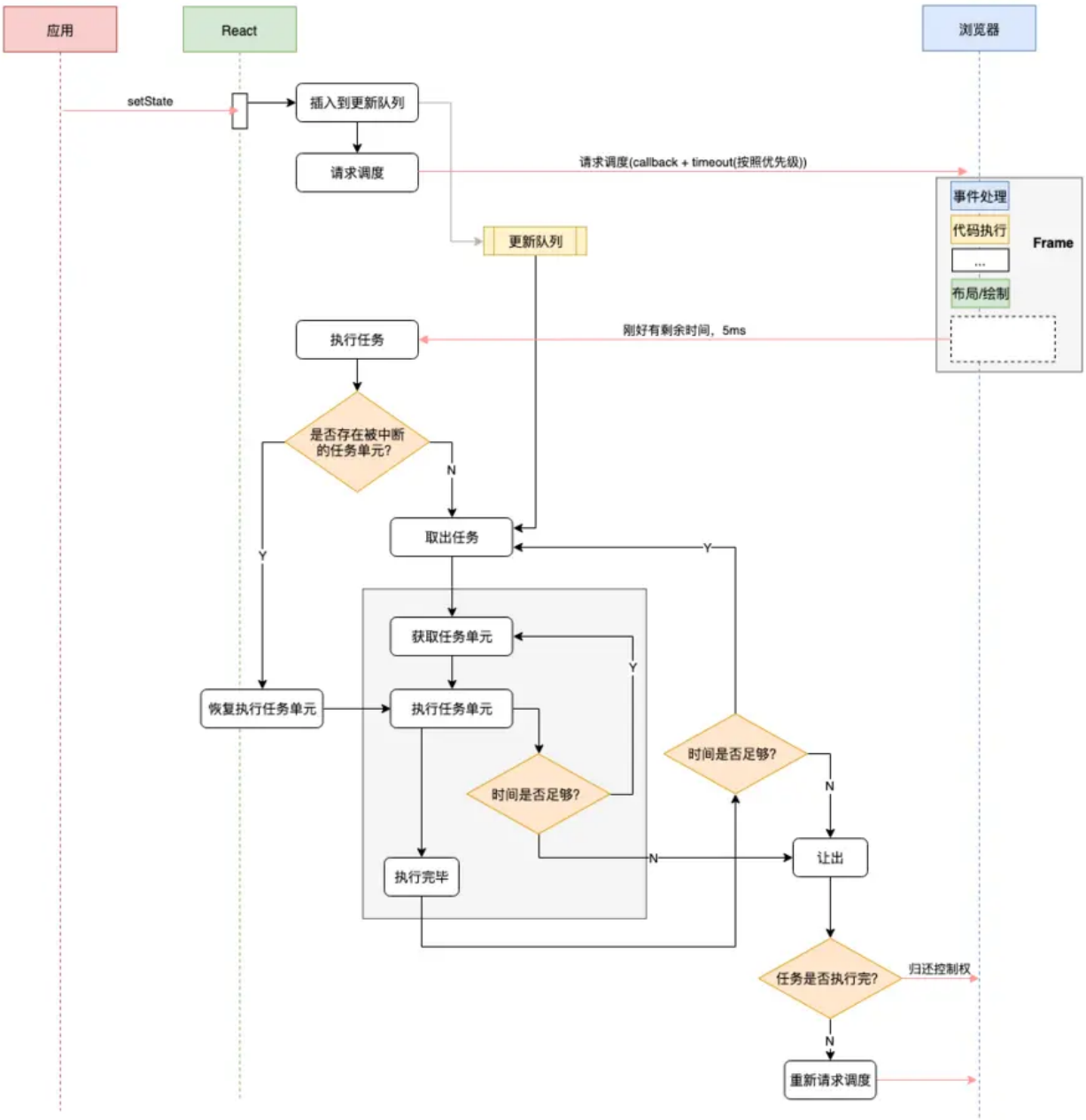
```js
// 1️⃣ performWork 会拿到一个Deadline，表示剩余时间
function performWork(deadline) {

  // 2️⃣ 循环取出updateQueue中的任务
  while (updateQueue.length > 0 && deadline.timeRemaining() > ENOUGH_TIME) {
    workLoop(deadline);
  }

  // 3️⃣ 如果在本次执行中，未能将所有任务执行完毕，那就再请求浏览器调度
  if (updateQueue.length > 0) {
    requestIdleCallback(performWork);
  }
}
```

`workLoop` 的工作大概猜到了，它会从更新队列(updateQueue)中弹出更新任务来执行，每执行完一个'`执行单元`'，就检查一下剩余时间是否充足，如果充足就进行执行下一个 `执行单元` ，反之则停止执行，保存现场，等下一次有执行权时恢复:

```js
// 保存当前的处理现场
let nextUnitOfWork: Fiber | undefined // 保存下一个需要处理的工作单元
let topWork: Fiber | undefined        // 保存第一个工作单元

function workLoop(deadline: IdleDeadline) {
  // updateQueue中获取下一个或者恢复上一次中断的执行单元
  if (nextUnitOfWork == null) {
    nextUnitOfWork = topWork = getNextUnitOfWork();
  }
```

```
  // 🔴 每执行完一个执行单元，检查一次剩余时间
  // 如果被中断，下一次执行还是从 nextUnitOfWork 开始处理
  while (nextUnitOfWork && deadline.timeRemaining() > ENOUGH_TIME) {
    // 下文我们再看performUnitOfWork
    nextUnitOfWork = performUnitOfWork(nextUnitOfWork, topWork);
  }

  // 提交工作，下文会介绍
  if (pendingCommit) {
    commitAllWork(pendingCommit);
  }
}
```

画个流程图吧！



React 的Fiber改造

Fiber 的核心内容已经介绍完了，现在来进一步看看React 为 Fiber 架构做了哪些改造, 如果你对这部分内容不感兴趣可以跳过。

1. 数据结构的调整

左侧是Virtual DOM，右侧可以看作diff的递归调用栈

上文中提到 React 16 之前，Reconcilation 是同步的、递归执行的。也就是说这是基于函数'调用栈'的Reconcilation算法，因此通常也称它为 `Stack Reconcilation` . 你可以通过这篇文章《从Preact中了解React组件和hooks基本原理》 来回顾一下历史。

栈挺好的，代码量少，递归容易理解, 至少比现在的 React Fiber架构好理解😂 , 递归非常适合树这种嵌套数据结构的处理。

只不过这种依赖于调用栈的方式不能随意中断、也很难被恢复, 不利于异步处理。 这种调用栈，不是程序所能控制的， 如果你要恢复递归现场，可能需要从头开始, 恢复到之前的调用栈。

因此**首先我们需要对React现有的数据结构进行调整， 模拟函数调用栈 , 将之前需要递归进行处理的事情分解成增量的执行单元，将递归转换成迭代**.

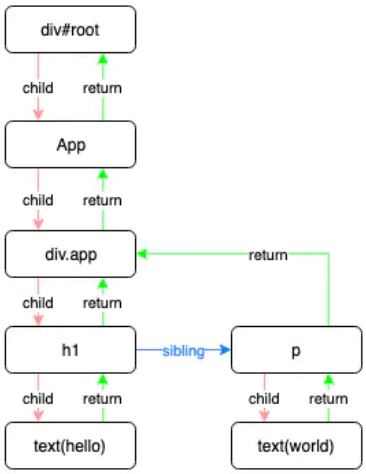
React 目前的做法是使用 链表 , 每个 VirtualDOM 节点内部现在使用 `Fiber` 表示, 它的结构大概如下:

```js
export type Fiber = {
  // Fiber 类型信息
  type: any,
  // ...

  // ⚛ 链表结构
  // 指向父节点，或者render该节点的组件
  return: Fiber | null,
  // 指向第一个子节点
  child: Fiber | null,
```

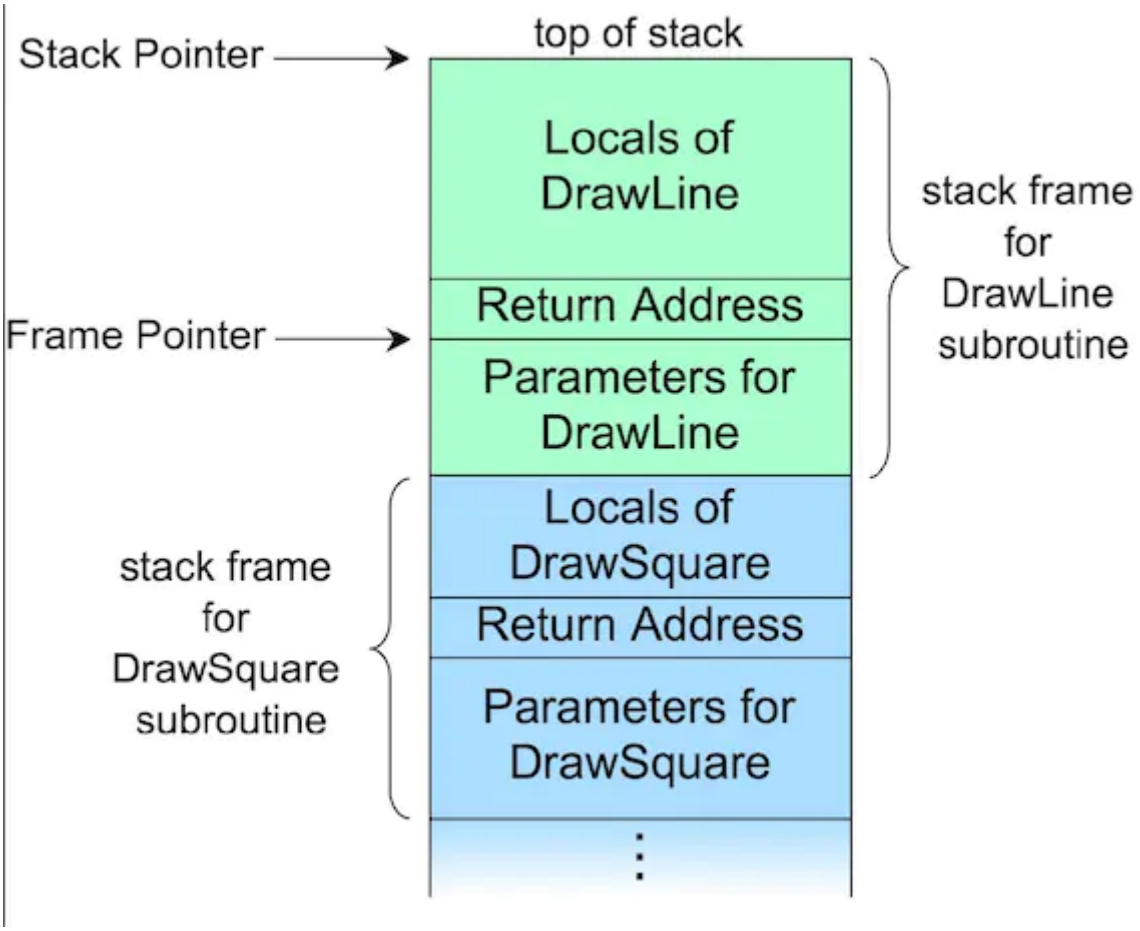
```
  // 指向下一个兄弟节点
  sibling: Fiber | null,
}
```

用图片来展示这种关系会更直观一些：

```
function App() {
  return <div class="app">
    <h1>Hello</h1>
    <p>world</p>
  </div>
}

render(<App/>, document.getElementById('root'))
```



使用链表结构只是一个结果，而不是目的，React 开发者一开始的目的是冲着模拟调用栈去的。这个很多关于Fiber 的文章都有提及, 关于调用栈的详细定义参见Wiki：



> 调用栈最经常被用于存放子程序的**返回地址**。在调用任何子程序时，主程序都必须暂存子程序运行完毕后应该返回到的地址。因此，如果被调用的子程序还要调用其他的子程序，其自身的返回地址就必须存入调用栈，在其自身运行完毕后再行取回。除了返回地址，还会保存 本地变量 、 函数参数 、 环境传递 (Scope?)

React Fiber 也被称为虚拟栈帧(Virtual Stack Frame), 你可以拿它和函数调用栈类比一下, 两者结构非常像:

	函数调用栈	Fiber
基本单位	函数	Virtual DOM 节点
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	嵌套函数调用	子节点(child)
上级引用	返回地址	父节点(return)

Fiber 和调用栈帧一样, 保存了节点处理的上下文信息，因为是手动实现的，所以更为可控，我们可以保存在内存中，随时中断和恢复。

有了这个数据结构调整，现在可以以迭代的方式来处理这些节点了。来看看 `performUnitOfWork` 的实现, 它其实就是一个深度优先的遍历：

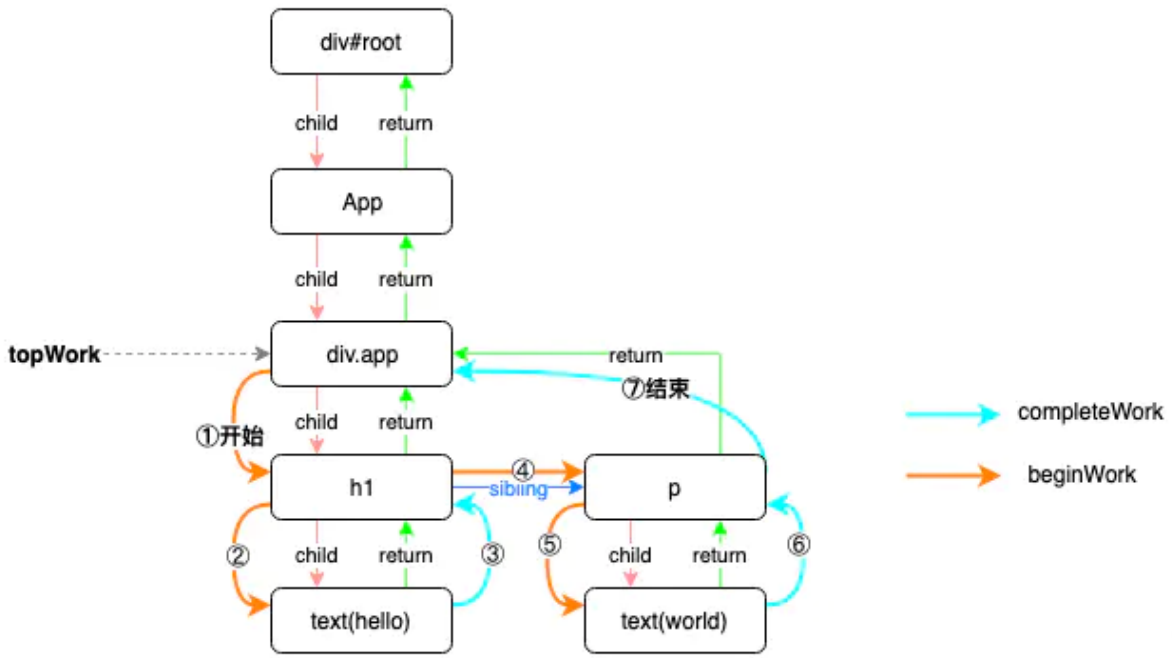
```js
/**
 * @params fiber 当前需要处理的节点
 * @params topWork 本次更新的根节点
 */
function performUnitOfWork(fiber: Fiber, topWork: Fiber) {
  // 对该节点进行处理
  beginWork(fiber);

  // 如果存在子节点，那么下一个待处理的就是子节点
  if (fiber.child) {
    return fiber.child;
  }

  // 没有子节点了，上溯查找兄弟节点
  let temp = fiber;
  while (temp) {
    completeWork(temp);

    // 到顶层节点了，退出
```

```
    if (temp === topWork) {
      break
    }

    // 找到，下一个要处理的就是兄弟节点
    if (temp.sibling) {
      return temp.sibling;
    }

    // 没有，继续上溯
    temp = temp.return;
  }
}
```

你可以配合上文的 `workLoop` 一起看，**Fiber 就是我们所说的工作单元，`performUnitOfWork` 负责对 `Fiber` 进行操作，并按照深度遍历的顺序返回下一个 Fiber。**

因为使用了链表结构，即使处理流程被中断了，我们随时可以从上次未处理完的 `Fiber` 继续遍历下去。

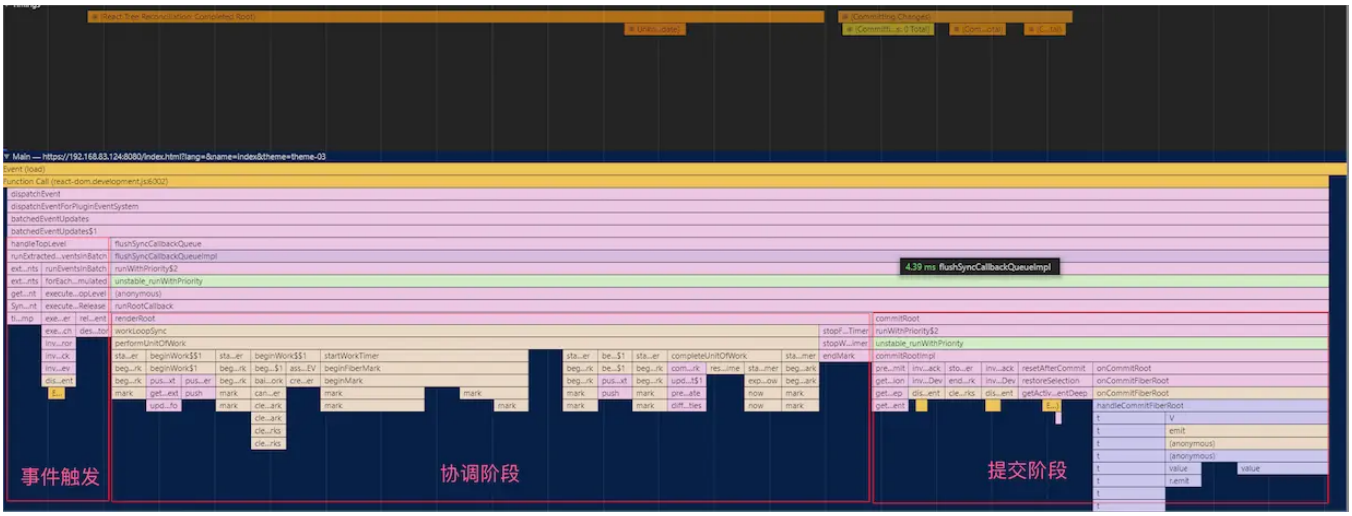
整个迭代顺序和之前递归的一样, 下图假设在 `div.app` 进行了更新：



比如你在 `text(hello)` 中断了，那么下一次就会从 `p` 节点开始处理

这个数据结构调整还有一个好处，就是某些节点异常时，我们可以打印出完整的'节点栈'，只需要沿着节点的 `return` 回溯即可。

2. 两个阶段的拆分



如果你现在使用最新的 React 版本(v16), 使用 Chrome 的 Performance 工具，可以很清晰地看到每次渲染有两个阶段： `Reconciliation` (协调阶段) 和 `Commit` (提交阶段).

> 我在之前的多篇文章中都有提及: 《自己写个React渲染器: 以 Remax 为例(用React写小程序)》

除了Fiber 工作单元的拆分，两阶段的拆分也是一个非常重要的改造，在此之前都是一边Diff一边提交的。先来看看这两者的区别:

- **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)' . 以下生命周期钩子会调阶段被调用:
  - `constructor`

    - `componentWillMount` 废弃
    - `componentWillReceiveProps` 废弃
    - `static getDerivedStateFromProps`
    - `shouldComponentUpdate`
    - `componentWillUpdate` 废弃
    - `render`

- ⚛️ **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**. 这些生命周期钩子在提交阶段被执行:

    - `getSnapshotBeforeUpdate()` 严格来说，这个是在进入 commit 阶段前调用
    - `componentDidMount`
    - `componentDidUpdate`
    - `componentWillUnmount`

也就是说，在协调阶段如果时间片用完，React就会选择让出控制权。因为协调阶段执行的工作不会导致任何用户可见的变更，所以在这个阶段让出控制权不会有什么问题。

需要注意的是：因为协调阶段可能被中断、恢复，甚至重做，⚠️ **React 协调阶段的生命周期钩子可能会被调用多次!**, 例如 `componentWillMount` 可能会被调用两次。

因此建议 **协调阶段的生命周期钩子不要包含副作用**. 索性 React 就废弃了这部分可能包含副作用的生命周期方法，例如 `componentWillMount` 、 `componentWillUpdate` . v17后我们就不能再用它们了, 所以现有的应用应该尽快迁移.

现在你应该知道为什么'提交阶段'必须同步执行，不能中断的吧？ 因为我们要正确地处理各种副作用，包括DOM变更、还有你在 `componentDidMount` 中发起的异步请求、useEffect 中定义的副作用... 因为有副作用，所以必须保证按照次序只调用一次，况且会有用户可以察觉到的变更, 不容差池。

关于为什么要拆分两个阶段，这里有更详细的解释。

3. Reconcilation

接下来就是就是我们熟知的 `Reconcilation` (为了方便理解，本文不区分Diff和Reconcilation, 两个同一个东西)阶段了. **思路和 Fiber 重构之前差别不大, 只不过这里不会再递归去比对、而且不会马上提**

交变更。

首先再进一步看一下 Fiber 的结构:

```ts
interface Fiber {
  /**
   * ⚛ 节点的类型信息
   */
  // 标记 Fiber 类型, 例如函数组件、类组件、宿主组件
  tag: WorkTag,
  // 节点元素类型, 是具体的类组件、函数组件、宿主组件(字符串)
  type: any,

  /**
   * ⚛ 结构信息
   */
  return: Fiber | null,
  child: Fiber | null,
  sibling: Fiber | null,
  // 子节点的唯一键, 即我们渲染列表传入的key属性
  key: null | string,

  /**
   * ⚛ 节点的状态
   */
  // 节点实例(状态):
  //         对于宿主组件, 这里保存宿主组件的实例, 例如DOM节点。
  //         对于类组件来说, 这里保存类组件的实例
  //         对于函数组件说, 这里为空, 因为函数组件没有实例
  stateNode: any,
  // 新的、待处理的props
  pendingProps: any,
  // 上一次渲染的props
  memoizedProps: any, // The props used to create the output.
  // 上一次渲染的组件状态
  memoizedState: any,


  /**
   * ⚛ 副作用
   */
  // 当前节点的副作用类型, 例如节点更新、删除、移动
  effectTag: SideEffectTag,
  // 和节点关系一样, React 同样使用链表来将所有有副作用的Fiber连接起来
  nextEffect: Fiber | null,

  /**
   * ⚛ 替身
```

```
    * 指向旧树中的节点
    */
   alternate: Fiber | null,
}
```

Fiber 包含的属性可以划分为 5 个部分:

- 🆕 **结构信息** - 这个上文我们已经见过了，Fiber 使用链表的形式来表示节点在树中的定位
- **节点类型信息** - 这个也容易理解，tag表示节点的分类、type 保存具体的类型值，如div、MyComp
- **节点的状态** - 节点的组件实例、props、state等，它们将影响组件的输出
- 🆕 **副作用** - 这个也是新东西. 在 Reconciliation 过程中发现的'副作用'(变更需求)就保存在节点的 `effectTag` 中(想象为打上一个标记). 那么怎么将本次渲染的所有节点副作用都收集起来呢？ 这里也使用了链表结构，在遍历过程中React会将所有有'副作用'的节点都通过 `nextEffect` 连接起来
- 🆕 **替身** - React 在 Reconciliation 过程中会构建一颗新的树(官方称为workInProgress tree，**WIP树**)，可以认为是一颗表示当前工作进度的树。还有一颗表示已渲染界面的**旧树**，React就是一边和旧树比对，一边构建WIP树的。 alternate 指向旧树的同等节点。

现在可以放大看看 `beginWork` 是如何对 Fiber 进行比对的:

```ts
function beginWork(fiber: Fiber): Fiber | undefined {
  if (fiber.tag === WorkTag.HostComponent) {
    // 宿主节点diff
    diffHostComponent(fiber)
  } else if (fiber.tag === WorkTag.ClassComponent) {
    // 类组件节点diff
    diffClassComponent(fiber)
  } else if (fiber.tag === WorkTag.FunctionComponent) {
    // 函数组件节点diff
    diffFunctionalComponent(fiber)
  } else {
    // ... 其他类型节点，省略
  }
}
```

宿主节点比对:

```ts
function diffHostComponent(fiber: Fiber) {
  // 新增节点
  if (fiber.stateNode == null) {
    fiber.stateNode = createHostComponent(fiber)
  } else {
    updateHostComponent(fiber)
  }

  const newChildren = fiber.pendingProps.children;

  // 比对子节点
  diffChildren(fiber, newChildren);
}
```

类组件节点比对也差不多:

```ts
function diffClassComponent(fiber: Fiber) {
  // 创建组件实例
  if (fiber.stateNode == null) {
    fiber.stateNode = createInstance(fiber);
  }

  if (fiber.hasMounted) {
    // 调用更新前生命周期钩子
    applybeforeUpdateHooks(fiber)
  } else {
    // 调用挂载前生命周期钩子
    applybeforeMountHooks(fiber)
  }

  // 渲染新节点
  const newChildren = fiber.stateNode.render();
  // 比对子节点
  diffChildren(fiber, newChildren);

  fiber.memoizedState = fiber.stateNode.state
}
```

子节点比对:

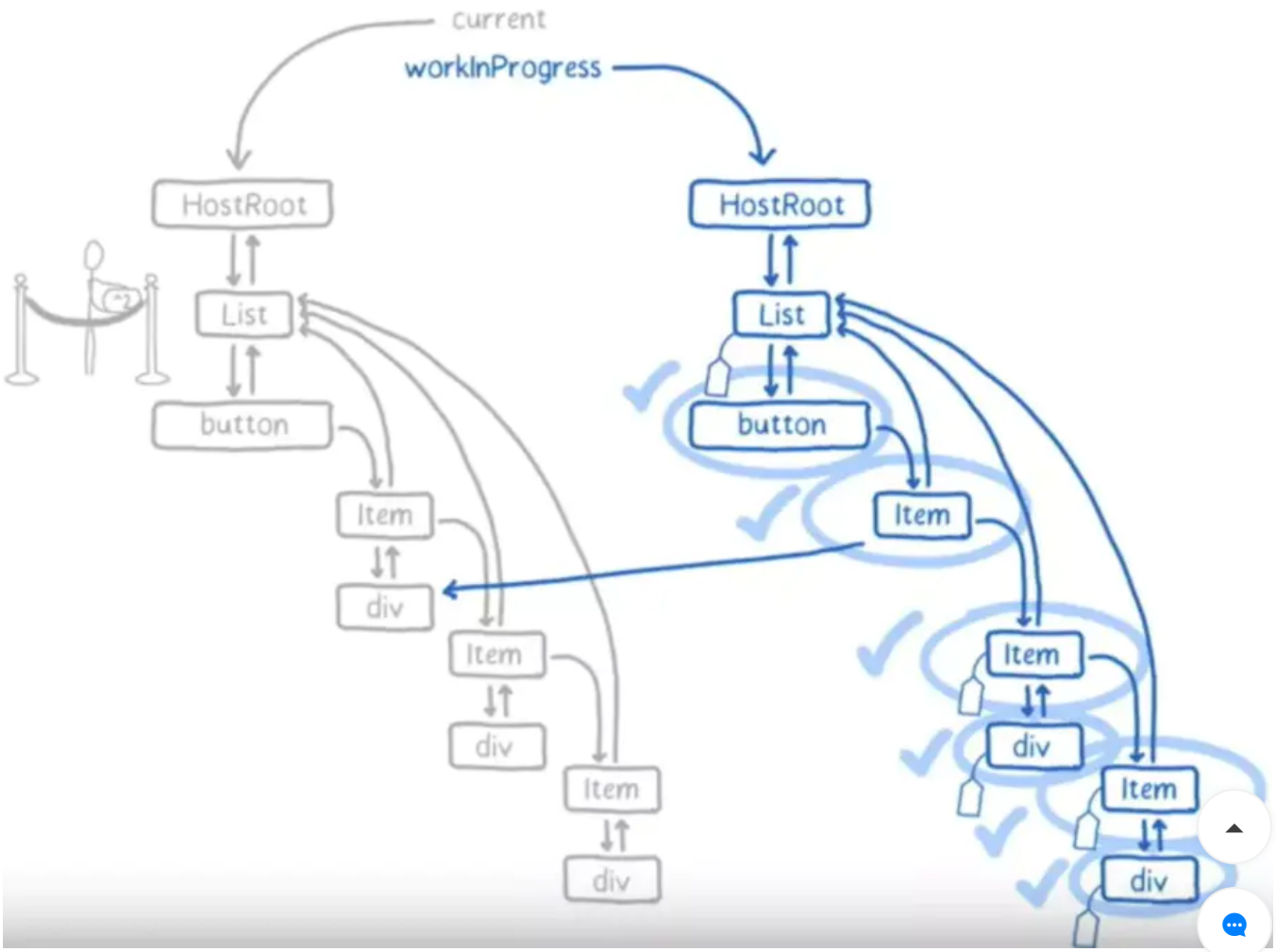
```ts
function diffChildren(fiber: Fiber, newChildren: React.ReactNode) {
  let oldFiber = fiber.alternate ? fiber.alternate.child : null;
  // 全新节点，直接挂载
  if (oldFiber == null) {
    mountChildFibers(fiber, newChildren)
    return
  }

  let index = 0;
  let newFiber = null;
  // 新子节点
  const elements = extraElements(newChildren)

  // 比对子元素
  while (index < elements.length || oldFiber != null) {
    const prevFiber = newFiber;
    const element = elements[index]
    const sameType = isSameType(element, oldFiber)
    if (sameType) {
      newFiber = cloneFiber(oldFiber, element)
      // 更新关系
      newFiber.alternate = oldFiber
      // 打上Tag
      newFiber.effectTag = UPDATE
      newFiber.return = fiber
    }

    // 新节点
    if (element && !sameType) {
      newFiber = createFiber(element)
      newFiber.effectTag = PLACEMENT
      newFiber.return = fiber
    }

    // 删除旧节点
    if (oldFiber && !sameType) {
      oldFiber.effectTag = DELETION;
      oldFiber.nextEffect = fiber.nextEffect
      fiber.nextEffect = oldFiber
    }

    if (oldFiber) {
      oldFiber = oldFiber.sibling;
    }

    if (index == 0) {
```

```
        fiber.child = newFiber;
      } else if (prevFiber && element) {
        prevFiber.sibling = newFiber;
      }

      index++
    }
  }
```

上面的代码很粗糙地还原了 Reconciliation 的过程, 但是对于我们理解React的基本原理已经足够了.

这里引用一下Youtube: Lin Clark presentation in ReactConf 2017 的Slide，来还原 Reconciliation 的过程. Lin Clark 这个演讲太经典了，几乎所有介绍 React Fiber 的文章都会引用它的Slide. 偷个懒，我也用下:

> 这篇文章《React Fiber》 用文字版解释了Link Clark Slide.



上图是 Reconciliation 完成后的状态，左边是旧树，右边是WIP树。对于需要变更的节点，都打上了'标签'。 在提交阶段，React 就会将这些打上标签的节点应用变更。

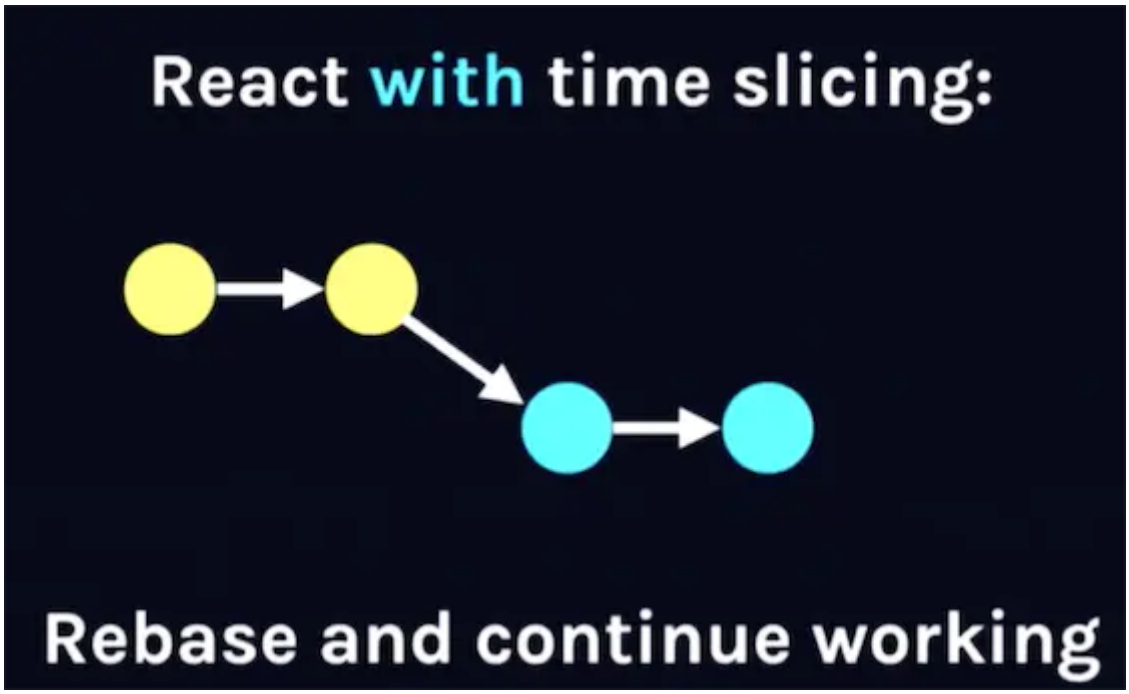
4. 双缓冲

`WIP 树` 构建这种技术类似于图形化领域的'**双缓存(Double Buffering)**'技术, 图形绘制引擎一般会使用双缓冲技术，先将图片绘制到一个缓冲区，再一次性传递给屏幕进行显示，这样可以防止屏幕抖动，优化渲染性能。

放到React 中，WIP树就是一个缓冲，它在Reconciliation 完毕后一次性提交给浏览器进行渲染。它可以减少内存分配和垃圾回收，WIP 的节点不完全是新的，比如某颗子树不需要变动，React会克隆复用旧树中的子树。

双缓存技术还有另外一个重要的场景就是异常的处理，比如当一个节点抛出异常，仍然可以继续沿用旧树的节点，避免整棵树挂掉。

Dan 在 Beyond React 16 演讲中用了一个非常恰当的比喻，那就是Git 功能分支，**你可以将 WIP 树想象成从旧树中 Fork 出来的功能分支，你在这新分支中添加或移除特性，即使是操作失误也不会影响旧的分支。当你这个分支经过了测试和完善，就可以合并到旧分支，将其替换掉. 这或许就是'提交(commit)阶段'的提交一词的来源吧？**：



5. 副作用的收集和提交

接下来就是将所有打了 Effect 标记的节点串联起来，这个可以在 `completeWork` 中做, 例如:

```ts
function completeWork(fiber) {
  const parent = fiber.return

  // 到达顶端
  if (parent == null || fiber === topWork) {
    pendingCommit = fiber
    return
  }

  if (fiber.effectTag != null) {
    if (parent.nextEffect) {
      parent.nextEffect.nextEffect = fiber
    } else {
      parent.nextEffect = fiber
    }
  } else if (fiber.nextEffect) {
    parent.nextEffect = fiber.nextEffect
  }
}
```

最后了，将所有副作用提交了：

```ts
function commitAllWork(fiber) {
  let next = fiber
  while(next) {
    if (fiber.effectTag) {
      // 提交，偷一下懒，这里就不展开了
      commitWork(fiber)
    }
    next = fiber.nextEffect
  }

  // 清理现场
  pendingCommit = nextUnitOfWork = topWork = null
}
```

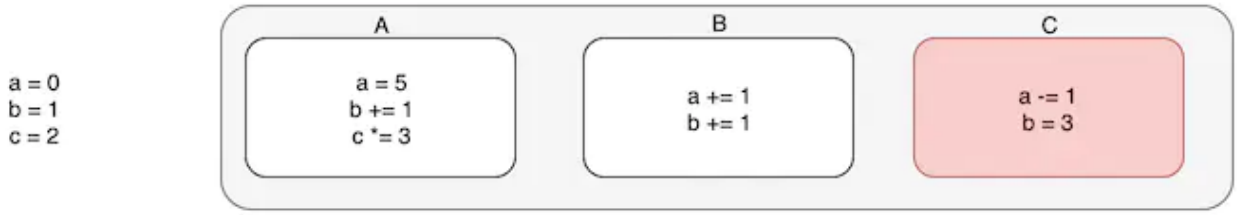
⚠️ 未展开部分🚧 -- 中断和恢复

上文只是介绍了简单的中断和恢复机制，我们从哪里跌倒就从哪里站起来，在哪个节点中断就从哪个节点继续处理下去。 也就是说，到目前为止：⚠️ **更新任务还是串行执行的，我们只是将整个过程碎片化了. 对于那些需要优先处理的更新任务还是会被阻塞**。我个人觉得这才是 React Fiber 中最难处理的一部分。

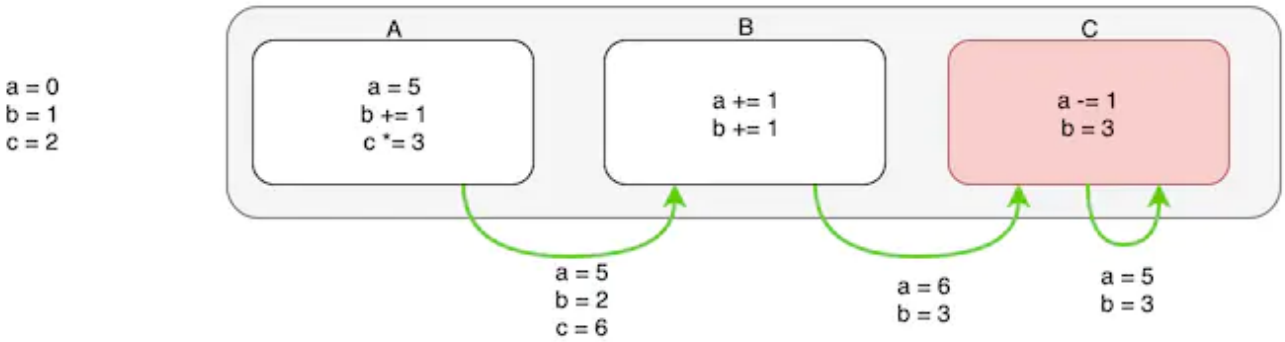
实际情况是，在 React 得到控制权后，应该优先处理高优先级的任务。也就是说中断时正在处理的任务，在恢复时会让位给高优先级任务，原本中断的任务可能会被放弃或者重做。

但是如果不按顺序执行任务，可能会导致前后的状态不一致。 比如低优先级任务将 `a` 设置为0，而高优先级任务将 `a` 递增1, 两个任务的执行顺序会影响最终的渲染结果。因此**要让高优先级任务插队, 首先要保证状态更新的时序**。

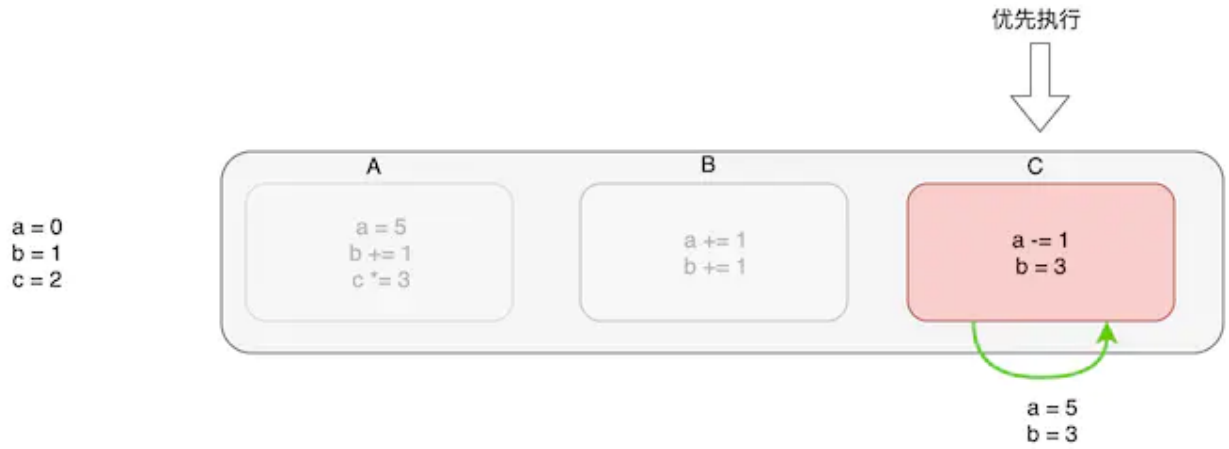
解决办法是: **所有更新任务按照顺序插入一个队列, 状态必须按照插入顺序进行计算，但任务可以按优先级顺序执行**, 例如：



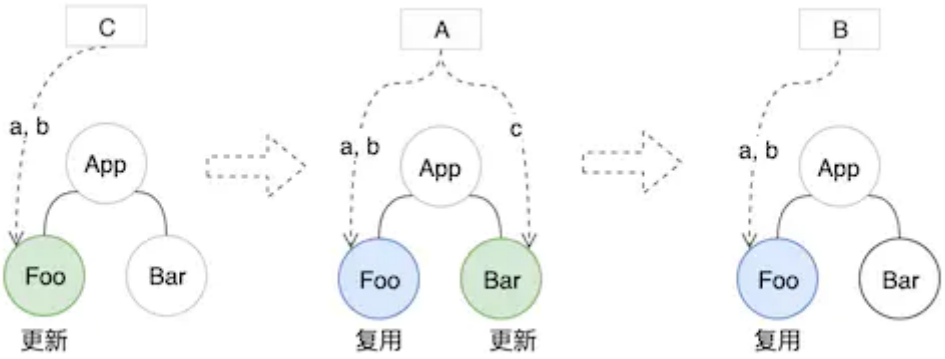
红色表示高优先级任务。要计算它的状态必须基于前序任务计算出来的状态, 从而保证**状态的最终一致性**:



最终红色的高优先级任务 `C` 执行时的状态值是 `a=5,b=3` . 在恢复控制权时，会按照优先级先执行 `C` , 前面的 `A` 、 `B` 暂时跳过



上面被跳过任务不会被移除，在执行完高优先级任务后它们还是会被执行的。因为不同的更新任务影响的节点树范围可能是不一样的，举个例子 `a` 、 `b` 可能会影响 `Foo` 组件树，而 `c` 会影响 `Bar` 组件树。所以为了保证**视图的最终一致性**, 所有更新任务都要被执行。



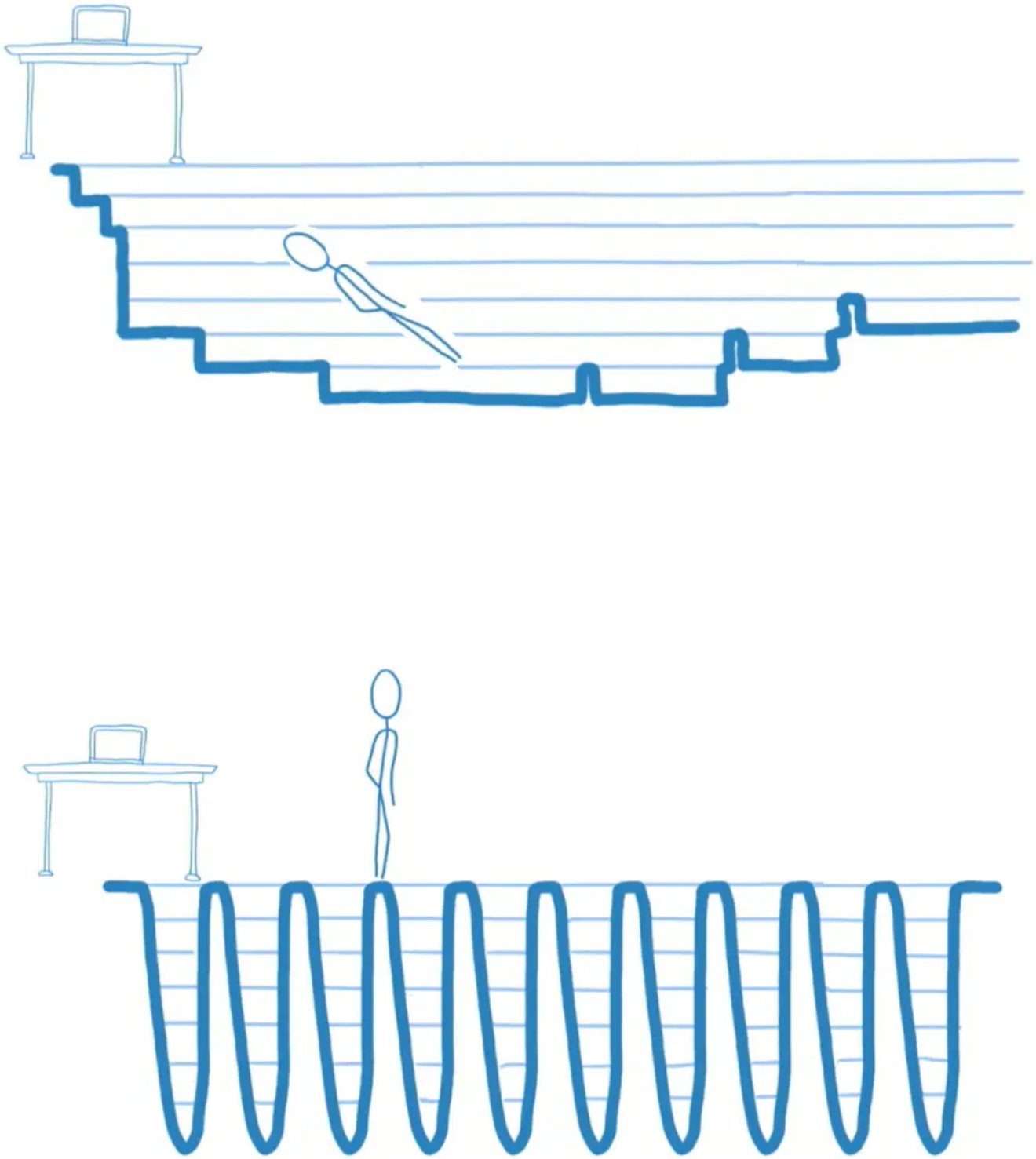
首先 `C` 先被执行，它更新了 `Foo` 组件

接着执行 `A` 任务，它更新了 `Foo` 和 `Bar` 组件，由于 `C` 已经以最终状态 `a=5, b=3` 更新了 `Foo` 组件，这里可以做一下性能优化，直接复用C的更新结果， 不必触发重新渲染。因此 `A` 仅需更新 `Bar` 组件即可。

接着执行 `B` ，同理可以复用 Foo 更新结果。

道理讲起来都很简单，React Fiber 实际上非常复杂，不管执行的过程怎样拆分、以什么顺序执行，最重要的是保证**状态的一致性**和**视图的一致性**，这给了 React 团队很大的考验，以致于现在都没有正式release出来。

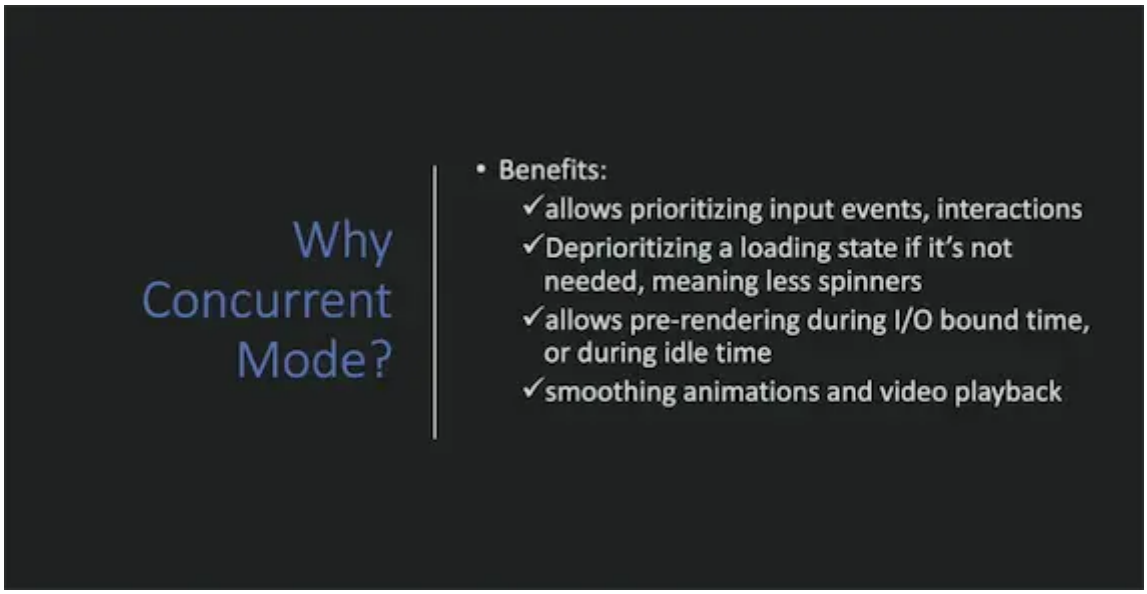
凌波微步



同样来自Link Clark 的 Slider

前面说了一大堆，从操作系统进程调度、到浏览器原理、再到合作式调度、最后谈了React的基本改造工作, 地老天荒... 就是为了上面的小人可以在练就凌波微步, 它脚下的坑是浏览器的调用栈。

React 开启 `Concurrent Mode` 之后就不会挖大坑了，而是一小坑一坑的挖，挖一下休息一下，有紧急任务就优先去做。



来源：Flarnie Marchan - Ready for Concurrent Mode?

开启 `Concurrent Mode` 后，我们可以得到以下好处(详见Concurrent Rendering in React):

- 快速响应用户操作和输入，提升用户交互体验
- 让动画更加流畅，通过调度，可以让应用保持高帧率
- 利用好I/O 操作空闲期或者CPU空闲期，进行一些预渲染。 比如离屏(offscreen)不可见的内容，优先级最低，可以让 React 等到CPU空闲时才去渲染这部分内容。这和浏览器的preload等预加载技术差不多。
- 用 `Suspense` 降低加载状态(load state)的优先级，减少闪屏。 比如数据很快返回时，可以不必显示加载状态，而是直接显示出来，避免闪屏；如果超时没有返回才显式加载状态。

但是它肯定不是完美的，因为浏览器无法实现抢占式调度，无法阻止开发者做傻事的，开发者可以随心所欲，想挖多大的坑，就挖多大的坑。

为了共同创造美好的世界，我们要严律于己，该做的优化还需要做: 纯组件、虚表、简化组件、缓存...

尤雨溪在今年的Vue Conf一个观点让我印象深刻：**如果我们可以把更新做得足够快的话，理论上就不需要时间分片了。**

时间分片并没有降低整体的工作量，该做的还是要做, 因此React 也在考虑利用CPU空闲或者I/O空闲做一些预渲染。所以跟尤雨溪说的一样：React Fiber 本质上是为了解决 React 更新低效率的问题，**不要期望 Fiber 能给你现有应用带来质的提升，如果性能问题是自己造成的，自己的锅还是得自己**

站在巨人的肩膀上

本文之所以能成文，离不开社区上优质的开源项目和资料。

迷你 Fiber 实现:

React 现在的代码库太复杂了! 而且一直在变动和推翻自己，Hax 在 《为什么社区里那些类 React 库至今没有选择实现 Fiber 架构？》 就开玩笑说: Fiber 性价比略低... 到了这个阶段，竞品太多，facebook 就搞一个 fiber 来作为护城河.....

这种工程量不是一般团队能Hold住的， 如果你只是想了解 Fiber，去读 React 的源码性价比也很低，不妨看看这些 Mini 版实现, 感受其精髓，不求甚解:

- anu 司徒正美 开发的类React框架
- Fre 伊撒尔 开发的类React框架，代码很精简!?
- Luy
- didact

优秀的文章 & 演讲

本文只是对React Fiber进行了简单的科普，实际上React 的实现比本文复杂的多，如果你想深入理解React Fiber的，下面这些文章不容错过:

- Lin Clark - A Cartoon Intro to Fiber - React Conf 2017 👍 📽️ React Fiber 启蒙，YouTube
- Beyond React 16 - Dan Abramov 👍 📽️
- Concurrent Rendering in React - Andrew Clark and Brian Vaughn 👍 📽️
- 司徒正美: React Fiber架构 👍 看不如写
- 展望 React 17，回顾 React 往事 👍 看完 Heaven 的相关文章，会觉得你了解的React 知识真的只是冰山一角，我们都没资格说我们懂 React。
- 浅入 React16/fiber 系列 👍 同样来自 Heaven
- 淡苍：深入剖析 React Concurrent 👍
- Didact Fiber: Incremental reconciliation 👍 实现了简单的 React Fiber
- 程墨: React Fiber是什么
- 译 深入React fiber架构及源码
- 黯羽轻扬: 完全理解React Fiber

- Fiber Principles: Contributing To Fiber
- Scheduling in React
- 桃翁: Deep In React 之浅谈 React Fiber 架构（一）
- 为 Luy 实现 React Fiber 架构
- 妖僧风月: React Fiber
- Flarnie Marchan - Ready for Concurrent Mode? 🎦
- Web Fundamentals > Performance
- 你应该知道的requestIdleCallback
- 深入探究 eventloop 与浏览器渲染的时序问题
- Accurately measuring layout on the web

自荐React 相关文章

回顾一下今年写的关于 React 的相关文章

Concurrent模式预览（推荐）：

- React Concurrent 模式抢先预览: Suspense 篇
- React Concurrent 模式抢先预览下篇: useTransition 的平行世界

往期文章:

- React组件设计实践总结 系列 共5篇
- 自己写个React渲染器: 以 Remax 为例(用React写小程序)
- 谈谈React事件机制和未来(react-events)
- 2019年了，整理了N个实用案例帮你快速迁移到React Hooks
- 浅谈React性能优化的方向
- 从Preact中了解React组件和hooks基本原理
- React性能测量和分析

本文讲了 React 如何优化 CPU 问题，React 野心远不在于此, I/O 方向的优化也在实践，例如 Suspend... 还有很多没讲完，后面的文章见！