阿里前端面试官带你手撕大厂秋招面试真题

1. JS：使用JS实现带并发的一部任务调度器

实现一个带并发限制的异步调度器 Scheduler，保证同时运行的任务最多有N个。完善下面代码中的 Scheduler 类，使得以下程序能正确输出：

```
class Scheduler {
  add(promiseCreator) { ... }
  // ...
}

const timeout = (time) => new Promise(resolve => {
  setTimeout(resolve, time)
})

const scheduler = new Scheduler(n)
const addTask = (time, order) => {
  scheduler.add(() => timeout(time)).then(() => console.log(order))
}

addTask(1000, '1')
addTask(500, '2')
addTask(300, '3')
addTask(400, '4')

// 打印顺序是: 2 3 1 4
```

1.1.1 流程分析

整个的完整执行流程：

> 起始1、2两个任务开始执行
> 500ms时，2任务执行完毕，输出2，任务3开始执行
> 800ms时，3任务执行完毕，输出3，任务4开始执行
> 1000ms时，1任务执行完毕，输出1，此时只剩下4任务在执行
> 1200ms时，4任务执行完毕，输出4

当资源不足时将任务加入等待队列，当资源足够时，将等待队列中的任务取出执行。

在调度器中一般会有一个等待队列queue，存放当资源不够时等待执行的任务
具有并发数据限制，假设通过max设置允许同时运行的任务，还需要count表示当前正在执行的任务数量
当需要执行一个任务A时，先判断count==max 如果相等说明任务A不能执行，应该被阻塞，阻塞的任务放进queue中，等待任务调度器管理。

如果count<max说明正在执行的任务数没有达到最大容量，那么count++执行任务A,执行完毕后count--
此时如果queue中有值，说明之前有任务因为并发数量限制而被阻塞，现在count<max，任务调度器会将对头的任务弹出执行。

```
class Scheduler {
  constructor(max) {
    this.max = max;
    this.count = 0; // 用来记录当前正在执行的异步函数
    this.queue = new Array(); // 表示等待队列
  }
  async add(promiseCreator) {
    /*
        此时count已经满了，不能执行本次add需要阻塞在这里，将resolve放入队列中等待唤醒，
        等到count<max时，从队列中取出执行resolve,执行，await执行完毕，本次add继续
        */
    if (this.count >= this.max) {
      await new Promise((resolve, reject) => this.queue.push(resolve));
    }

    this.count++;
    let res = await promiseCreator();
    this.count--;
    if (this.queue.length) {
      // 依次唤醒add
      // 若队列中有值，将其resolve弹出，并执行
      // 以便阻塞的任务，可以正常执行
      this.queue.shift()();
    }
    return res;
  }
}

const timeout = time =>
  new Promise(resolve => {
    setTimeout(resolve, time);
  });
```

```
33
34 const scheduler = new Scheduler(2);
35
36 const addTask = (time, order) => {
37   //add返回一个promise，参数也是一个promise
38   scheduler.add(() => timeout(time)).then(() => console.log(order));
39 };
40
41   addTask(1000, '1');
42   addTask(500, '2');
43   addTask(300, '3');
44   addTask(400, '4');
45
46 // output: 2 3 1 4
```

2. React：React fiber

什么是fiber?

React V15 在渲染时，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。这个过程期间， React 会占据浏览器资源，这会导致用户触发的事件得不到响应，并且会导致掉帧，导致用户感觉到卡顿。

为了给用户制造一种应用很快的“假象”，不能让一个任务长期霸占着资源。 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”，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，让这个执行过程变成可被中断。“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:
1. 分批延时对DOM进行操作，避免一次性操作大量 DOM 节点，可以得到更好的用户体验；

2. 给浏览器一点喘息的机会，它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。

核心思想：Fiber 也称协程或者纤程。它和线程并不一样，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。让出 CPU 的执行权，让 CPU 能在这段时间执行其他的操作。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。

如果你只能大概说出fiber的定义，甚至不知道什么是fiber，那么你的薪资大概率是很难突破20K的；如果你能回答到异步可中断，那么你基本上具备了P6左右的水平；如果你能把原理说清楚，在这个知

识点上，我认为你已经理解的还是比较透彻了，基本上对标P6+的水平了；

2.1 背景

1. React支持JSX语法，我们可以直接将HTML代码写到JS中间，然后渲染到页面上，我们写的HTML如果有更新的话，React还有虚拟DOM的对比，只更新变化的部分，而不重新渲染整个页面，大大提高渲染效率；
2. 到了V16.x，React更是使用了一个被称为Fiber的架构，提升了用户体验，同时还引入了hooks等特性。接下来从JSX入手，手写简单的React，了解其中原理；

2.2 JSX和CreateElement

我们知道，在实现React要支持JSX还需要一个库叫JSXTransformer.js，后来JSX的转换工作都集成到了babel里面了，babel还提供了在线预览的功能，可以看到转换后的效果：

```
const App =
  (
    <div>
      <h1 id="title">Title</h1>
      <a href="xxx">Jump</a>
      <section>
        <p>
          Article
        </p>
      </section>
    </div>
  );
```

转换后：

```
"use strict";

const App = /*#__PURE__*/React.createElement("div", null, /*#__PURE__*/React.cre
  id: "title"
}, "Title"), /*#__PURE__*/React.createElement("a", {
  href: "xxx"
}, "Jump"), /*#__PURE__*/React.createElement("section", null, /*#__PURE__*/React
```

格式化后为：

```
var App = React.createElement(
```

```
  'div',
  null,
  React.createElement(
    'h1',
    {
      id: 'title',
    },
    'Title',
  ),
  React.createElement(
    'a',
    {
      href: 'xxx',
    },
    'Jump',
  ),
  React.createElement(
    'section',
    null,
    React.createElement('p', null, 'Article'),
  ),

);
```

所以从转换后的代码，可以看到React.createElement支持以下几个参数：

1. type，也就是节点类型；
2. config, 这是节点上的属性，比如id和href；
3. children, 从第三个参数开始就全部是children也就是子元素了，子元素可以有多个，类型可以是简单的文本，也可以还是React.createElement，如果是React.createElement，其实就是子节点了，子节点下面还可以有子节点。这样就用React.createElement的嵌套关系实现了HTML节点的树形结构；

一个正常的React应用，最基础的除了createElement外，还有ReactDom.render，像下面：

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

const App = (
        <div>
                <h1 id="title">Title</h1>
                <section>
                        <p>Hello xianzao</p>
                </section>
```

```
        </div>
);

ReactDOM.render(App, document.getElementById('root'));
```

接下来，实现简单的React；

2.3 手写CreateElement

对于 `<h1 id="title">Title</h1>` 这样一个简单的节点，原生DOM也会附加一大堆属性和方法在上面，所以我们在createElement的时候最好能将它转换为一种比较简单的数据结构，只包含我们需要的元素：

```
{
  type: 'h1',
  props: {
    id: 'title',
    children: 'Title'
  }
}
```

有了这个数据结构后，我们对于DOM的操作其实可以转化为对这个数据结构的操作，新老DOM的对比其实也可以转化为这个数据结构的对比，这样我们就不需要每次操作都去渲染页面，而是等到需要渲染的时候才将这个数据结构渲染到页面上。这其实就是虚拟DOM！而我们createElement就是负责来构建这个虚拟DOM的方法：

```
function createElement(type, props, ...children) {
  // 核心逻辑不复杂，将参数都塞到一个对象上返回就行
  // children也要放到props里面去，这样我们在组件里面就能通过this.props.children拿到子
  return {
    type,
    props: {
      ...props,
      children
    }
  }
}
```

源码要比上述复杂的多，有兴趣可以看：
https://github.com/facebook/react/blob/60016c448bb7d19fc989acd05dda5aca2e124381/packages/react/src/ReactElement.js#L348

2.4 手写Render

我们用 `createElement` 将JSX代码转换成了虚拟DOM，那真正将它渲染到页面的函数是render，所以我们还需要实现下这个方法，通过我们一般的用法 `ReactDOM.render( <App />,document.getElementById('root'))` ：

1. 根组件，其实是一个JSX组件，也就是一个createElement返回的虚拟DOM；
2. 父节点，也就是我们要将这个虚拟DOM渲染的位置；

有了这两个参数，我们来实现下render方法：

```
function render(vDom, container) {
  let dom;
  // 检查当前节点是文本还是对象
  if(typeof vDom !== 'object') {
    dom = document.createTextNode(vDom)
  } else {
    dom = document.createElement(vDom.type);
  }

  // 将vDom上除了children外的属性都挂载到真正的DOM上去
  if(vDom.props) {
    Object.keys(vDom.props)
      .filter(key => key != 'children')
      .forEach(item => {
        dom[item] = vDom.props[item];
      })
  }

  // 如果还有子元素，递归调用
  if(vDom.props && vDom.props.children && vDom.props.children.length) {
    vDom.props.children.forEach(child => render(child, dom));
  }

  container.appendChild(dom);
}
```

源码地址：
https://github.com/facebook/react/blob/3e94bce765d355d74f6a60feb4addb6d196e3482/packages/react-dom/src/client/ReactDOMLegacy.js#L287

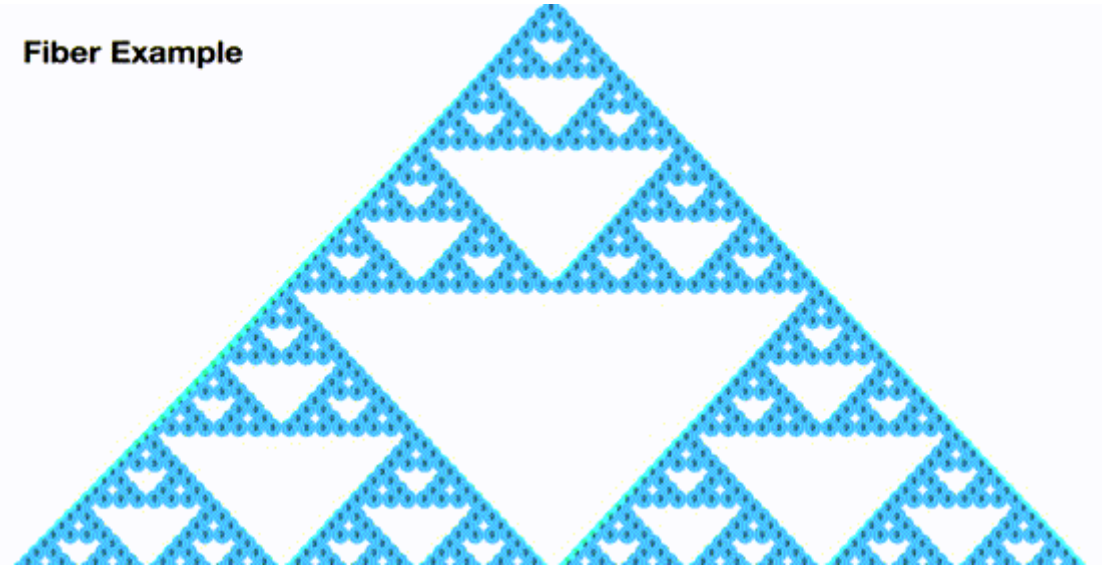
```
import React from './myReact';
```

```
const ReactDOM = React;

const App = (
        <div>
                <h1 id="title">Title</h1>
                <section>
                        <p>Hello xianzao</p>
                </section>
        </div>
);

ReactDOM.render(App, document.getElementById('root'));
```

2.5 为什么需要Fiber

1. 上述简单的实现了虚拟DOM渲染到页面上的代码，这部分工作被React官方称为 `renderer` ；
2. `renderer` 是第三方可以自己实现的一个模块，还有个核心模块叫做 `reconciler` ，`reconciler` 的一大功能就是大家熟知的 `diff` ，他会计算出应该更新哪些页面节点，然后将需要更新的节点虚拟DOM传递给 `renderer` ， `renderer` 负责将这些节点渲染到页面上；
3. 虽然React的 `diff` 算法是经过优化的，但是他却是同步的， `renderer` 负责操作DOM的 `appendChild` 等API也是同步的，也就是说如果有大量节点需要更新，JS线程的运行时间可能会比较长，在这段时间浏览器是不会响应其他事件的，因为JS线程和GUI线程是互斥的，JS运行时页面就不会响应，这个时间太长了，用户就可能看到卡顿，特别是动画的卡顿会很明显。在React的官方演讲中有个例子，可以很明显的看到这种同步计算造成的卡顿：

而Fiber就是用来解决这个问题的，Fiber可以将长时间的同步任务拆分成多个小任务，从而让浏览器能够抽身去响应其他事件，等他空了再回来继续计算，这样整个计算流程就显得平滑很多。下面是使用Fiber后的效果：



现在的问题：

1. 上面我们自己实现的 `render` 方法直接递归遍历了整个vDom树，如果我们在中途某一步停下来，下次再调用时其实并不知道上次在哪里停下来的，不知道从哪里开始，所以 `vDom` 的树形结构并不满足中途暂停，下次继续的需求，需要改造数据结构；
2. 拆分下来的小任务什么时候执行？我们的目的是让用户有更流畅的体验，所以我们最好不要阻塞高优先级的任务，比如用户输入，动画之类，等他们执行完了我们再计算。那我怎么知道现在有没有高优先级任务，浏览器是不是空闲呢？

总结下来， `Fiber` 要想达到目的，需要解决两个问题：

1. 新的任务调度，有高优先级任务的时候将浏览器让出来，等浏览器空了再继续执行；
2. 新的数据结构，可以随时中断，下次进来可以接着执行；

2.6 requestIdleCallback

`requestIdleCallback` 是一个实验中的新API，这个API调用方式如下:

- caniuse查看适配程度

```
// 开启调用
var handle = window.requestIdleCallback(callback[, options])

// 结束调用
Window.cancelIdleCallback(handle)
```

1. `requestIdleCallback` 接收一个回调，这个回调会在浏览器空闲时调用，每次调用会传入一个IdleDeadline，可以拿到当前还空余多久，options可以传入参数最多等多久，等到了时间浏览器还不空就强制执行了，使用这个API可以解决任务调度的问题，让浏览器在空闲时才计算diff并渲染；
2. 但是这个API还在实验中，兼容性不好，所以React官方自己实现了一套。（基于时间原因，本文会继续使用 `requestIdleCallback` 来进行任务调度）我们进行任务调度的思想是将任务拆分成多个小任务， `requestIdleCallback` 里面不断的把小任务拿出来执行，当所有任务都执行完或者超时了就结束本次执行，同时要注册下次执行，代码架子就是这样：

```
function workLoop(deadline) {
        while (nextUnitOfWork && deadline.timeRemaining() > 1) {
                // 这个while循环会在任务执行完或者时间到了的时候结束
                nextUnitOfWork = performUnitOfWork(nextUnitOfWork);
        }

        // 如果任务还没完，但是时间到了，我们需要继续注册requestIdleCallback
        requestIdleCallback(workLoop);
}

// performUnitOfWork用来执行任务，参数是我们的当前fiber任务，返回值是下一个任务
function performUnitOfWork(fiber) {}

// 使用requestIdleCallback开启workLoop
requestIdleCallback(workLoop);
```

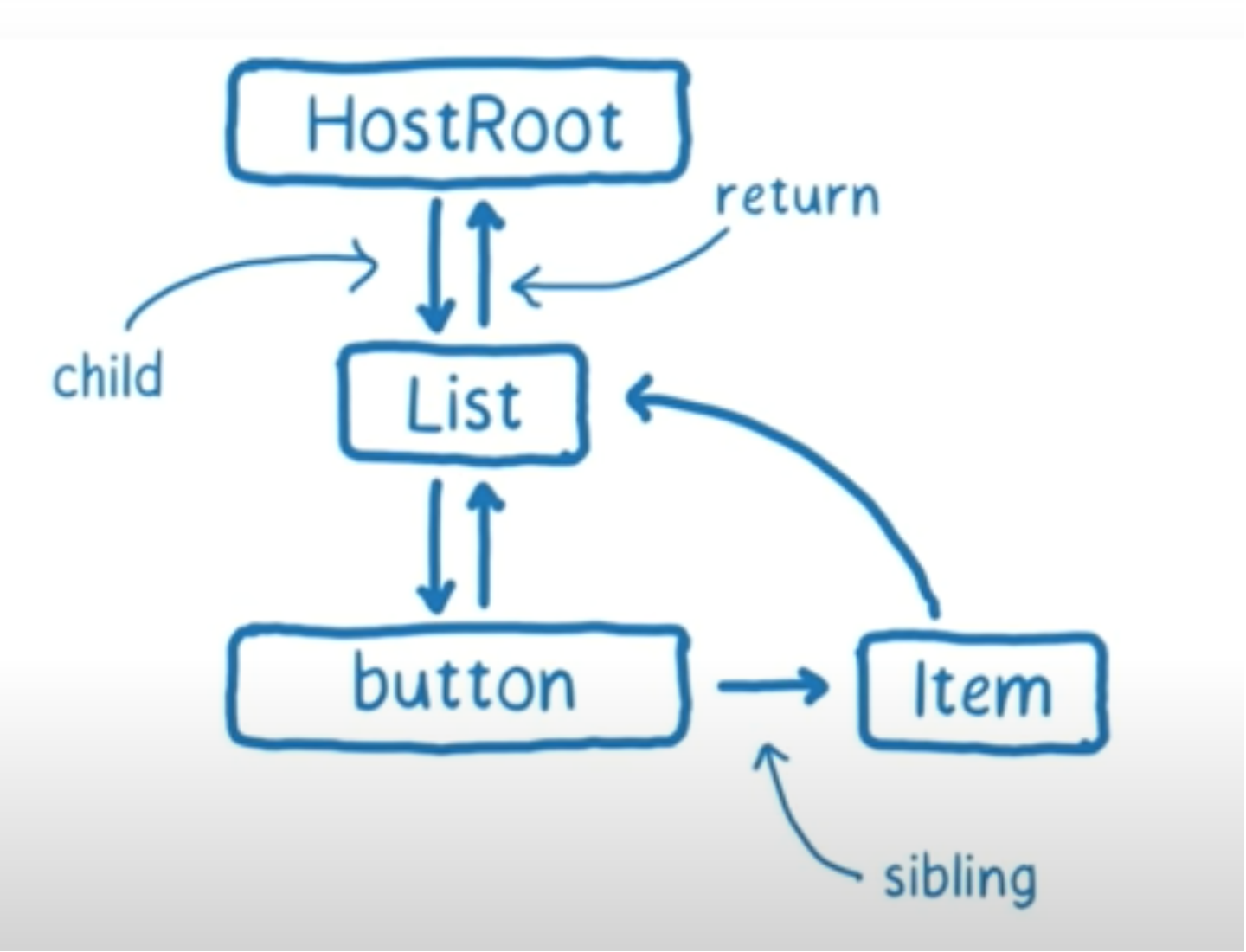
源码地址：
https://github.com/facebook/react/blob/4c7036e807fa18a3e21a5182983c7c0f05c5936e/packages/react-reconciler/src/ReactFiberWorkLoop.new.js#L1481

2.7 Fiber的可中断数据结构

接下来实现 `performUnitOfWork` ，从上面的结构可以看出来，他接收的参数是一个小任务，同时通过这个小任务还可以找到他的下一个小任务，Fiber构建的就是这样一个数据结构。

Fiber的数据结构是一棵树，包含3部分

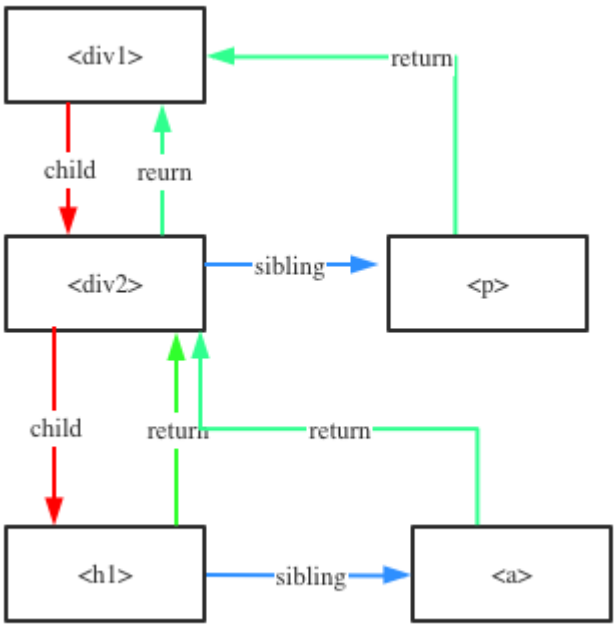
1. child: 父节点指向第一个子元素的指针；
2. sibling：从第一个子元素往后，指向下一个兄弟元素；
3. return：所有子元素都有的指向父元素的指针；



有了这几个指针后，我们可以在任意一个元素中断遍历并恢复，比如在上图 `List` 处中断了，恢复的时候可以通过 `child` 找到他的子元素，也可以通过 `return` 找到他的父元素，如果他还有兄弟节点也可以用 `sibling` 找到。Fiber这个结构外形看着还是棵树，但是没有了指向所有子元素的指针，父节点只指向第一个子节点，然后子节点有指向其他子节点的指针，这其实是个链表。

2.8 实现Fiber

1. 将之前的 `vDom` 结构转换为 `Fiber` 的数据结构，同时需要能够通过其中任意一个节点返回下一个节点，其实就是遍历这个链表；
2. 遍历的时候从根节点出发，先找子元素，如果子元素存在，直接返回，如果没有子元素了就找兄弟元素，找完所有的兄弟元素后再返回父元素，然后再找这个父元素的兄弟元素。整个遍历过程其实是个深度优先遍历（DFS），从上到下，然后最后一行开始从左到右遍历；
3. 比如下图从div1开始遍历的话，遍历的顺序就应该是 `div1 -> div2 -> h1 -> a -> div2 -> p -> div1` 。可以看到这个序列中，当我们return父节点时，这些父节点会被第二次遍历，所以我们写代码时，return的父节点不会作为下一个任务返回，只有 `sibling` 和 `child` 才会作为下一个任务返回；




```
// performUnitOfWork用来执行任务，参数是我们的当前fiber任务，返回值是下一个任务
function performUnitOfWork(fiber) {
  // 根节点的dom就是container，如果没有这个属性，说明当前fiber不是根节点
  if(!fiber.dom) {
    fiber.dom = createDom(fiber);    // 创建一个DOM挂载上去
  }

  // 如果有父节点，将当前节点挂载到父节点上
  if(fiber.return) {
    fiber.return.dom.appendChild(fiber.dom);
  }

  // 将我们前面的vDom结构转换为fiber结构
  const elements = fiber.children;
  let prevSibling = null;
  if(elements && elements.length) {
    for(let i = 0; i < elements.length; i++) {
      const element = elements[i];
      const newFiber = {
        type: element.type,
        props: element.props,
        return: fiber,
        dom: null
      }

      // 父级的child指向第一个子元素
      if(i === 0) {
        fiber.child = newFiber;
      } else {
```

```
        // 每个子元素拥有指向下一个子元素的指针
        prevSibling.sibling = newFiber;
      }

      prevSibling = newFiber;
    }
  }

  // 这个函数的返回值是下一个任务，这其实是一个深度优先遍历
  // 先找子元素，没有子元素了就找兄弟元素
  // 兄弟元素也没有了就返回父元素
  // 然后再找这个父元素的兄弟元素
  // 最后到根节点结束
  // 这个遍历的顺序其实就是从上到下，从左到右
  if(fiber.child) {
    return fiber.child;
  }

  let nextFiber = fiber;
  while(nextFiber) {
    if(nextFiber.sibling) {
      return nextFiber.sibling;
    }

    nextFiber = nextFiber.return;
  }
}
```

2.9 统一commit DOM操作

`performUnitOfWork` 一边构建Fiber结构一边操作 `DOMappendChild`，这样如果某次更新好几个节点，操作了第一个节点之后就中断了，那我们可能只看到第一个节点渲染到了页面，后续几个节点等浏览器空了才陆续渲染。为了避免这种情况，我们应该将DOM操作都搜集起来，最后统一执行，这就是 `commit`。为了能够记录位置，我们还需要一个全局变量 `workInProgressRoot` 来记录根节点，然后在 `workLoop` 检测如果任务执行完了，就 `commit`，

```
let workInProgressRoot = null; // 指向Fiber的根节点

function workLoop(deadline) {
  while(nextUnitOfWork && deadline.timeRemaining() > 1) {
    // 这个while循环会在任务执行完或者时间到了的时候结束
    nextUnitOfWork = performUnitOfWork(nextUnitOfWork);
  }

```

```
  // 任务做完后统一渲染
  if(!nextUnitOfWork && workInProgressRoot) {
    commitRoot();
  }

  // 如果任务还没完，但是时间到了，我们需要继续注册requestIdleCallback
  requestIdleCallback(workLoop);
}
```

因为我们是在Fiber树完全构建后再执行的 `commit` ，而且有一个变量 `workInProgressRoot` 指向了Fiber的根节点，所以我们可以直接把 `workInProgressRoot` 拿过来递归渲染就行了：

```
// 统一操作DOM
function commitRoot() {
  commitRootImpl(workInProgressRoot.child);    // 开启递归
  workInProgressRoot = null;      // 操作完后将workInProgressRoot重置
}

function commitRootImpl(fiber) {
  if(!fiber) {
    return;
  }

  const parentDom = fiber.return.dom;
  parentDom.appendChild(fiber.dom);

  // 递归操作子元素和兄弟元素
  commitRootImpl(fiber.child);
  commitRootImpl(fiber.sibling);
}
```

2.10 reconcile

`reconcile` 其实就是虚拟DOM树的 `diff` 操作：

- 删除不需要的节点；
- 更新修改过的节点；
- 添加新的节点；

为了在中断后能回到工作位置，我们还需要一个变量 `currentRoot` ，然后在fiber节点里面添加一个属性 `alternate` ，这个属性指向上一次运行的根节点，也就是 `currentRoot` 。 `currentRoot` 会在第一次 `render` 后的 `commit` 阶段赋值，也就是每次计算完后都会把当次状态记录在 `alternate` 上，后面更新了就可以把 `alternate` 拿出来跟新的状态做diff。然后

`performUnitOfWork` 里面需要添加调和子元素的代码，可以新增一个函数 `reconcileChildren`。这个函数要将老节点跟新节点拿来对比，对比逻辑如下:

1. 如果新老节点类型一样，复用老节点DOM，更新props;
2. 如果类型不一样，而且新的节点存在，创建新节点替换老节点;
3. 如果类型不一样，没有新节点，有老节点，删除老节点;

注意删除老节点的操作是直接将 `oldFiber` 加上一个删除标记就行，同时用一个全局变量deletions记录所有需要删除的节点:

```
// 对比oldFiber和当前element
const sameType = oldFiber && element && oldFiber.type === element.type;  //检测类
// 先比较元素类型
if(sameType) {
  // 如果类型一样，复用节点，更新props
  newFiber = {
    type: oldFiber.type,
    props: element.props,
    dom: oldFiber.dom,
    return: workInProgressFiber,
    alternate: oldFiber,          // 记录下上次状态
    effectTag: 'UPDATE'           // 添加一个操作标记
  }
} else if(!sameType && element) {
  // 如果类型不一样，有新的节点，创建新节点替换老节点
  newFiber = {
    type: element.type,
    props: element.props,
    dom: null,                    // 构建fiber时没有dom，下次perform这个节点是才创建
    return: workInProgressFiber,
    alternate: null,              // 新增的没有老状态
    effectTag: 'REPLACEMENT'      // 添加一个操作标记
  }
} else if(!sameType && oldFiber) {
  // 如果类型不一样，没有新节点，有老节点，删除老节点
  oldFiber.effectTag = 'DELETION';   // 添加删除标记
  deletions.push(oldFiber);          // 一个数组收集所有需要删除的节点
}
```

然后就是在 `commit` 阶段处理真正的DOM操作，具体的操作是根据我们的 `effectTag` 来判断的:

```
function commitRootImpl(fiber) {
  if(!fiber) {
    return;
```

```
  }

  const parentDom = fiber.return.dom;
  if(fiber.effectTag === 'REPLACEMENT' && fiber.dom) {
    parentDom.appendChild(fiber.dom);
  } else if(fiber.effectTag === 'DELETION') {
    parentDom.removeChild(fiber.dom);
  } else if(fiber.effectTag === 'UPDATE' && fiber.dom) {
    // 更新DOM属性
    updateDom(fiber.dom, fiber.alternate.props, fiber.props);
  }

  // 递归操作子元素和兄弟元素
  commitRootImpl(fiber.child);
  commitRootImpl(fiber.sibling);
}
```

替换和删除的DOM操作都比较简单，更新属性的会稍微麻烦点，我们用一个辅助函数 `updateDom` 来实现：

```
// 更新DOM的操作
function updateDom(dom, prevProps, nextProps) {
  // 1. 过滤children属性
  // 2. 老的存在，新的没了，取消
  // 3. 新的存在，老的没有，新增
  Object.keys(prevProps)
    .filter(name => name !== 'children')
    .filter(name => !(name in nextProps))
    .forEach(name => {
      if(name.indexOf('on') === 0) {
        dom.removeEventListener(name.substr(2).toLowerCase(), prevProps[name], f
      } else {
        dom[name] = '';
      }
    });

  Object.keys(nextProps)
    .filter(name => name !== 'children')
    .forEach(name => {
      if(name.indexOf('on') === 0) {
        dom.addEventListener(name.substr(2).toLowerCase(), nextProps[name], fals
      } else {
        dom[name] = nextProps[name];
      }
    });
```

```
26 }
```

3. 一个标准的前端所需要掌握的完整技术框架

印客2023大厂前端工程师课程大纲.pdf

4. Vue：Vue3响应式原理

- index.ts

```
1  export {
2  reactive,
3    readonly,
4    shallowReadonly,
5    isReadonly,
6    isReactive,
7    isProxy,
8  } from "./reactive";
9
10 export { ref, proxyRefs, unRef, isRef } from "./ref";
11
12 export { effect, stop, ReactiveEffect } from "./effect";
13
14 export { computed } from "./computed";
```

- reactive.ts

```ts
import {
  mutableHandlers,
  readonlyHandlers,
  shallowReadonlyHandlers,
} from "./baseHandlers";

export const reactiveMap = new WeakMap();
export const readonlyMap = new WeakMap();
export const shallowReadonlyMap = new WeakMap();

export const enum ReactiveFlags {
  IS_REACTIVE = "__v_isReactive",
  IS_READONLY = "__v_isReadonly",
  RAW = "__v_raw",
}

export function reactive(target) {
  return createReactiveObject(target, reactiveMap, mutableHandlers);
}

export function readonly(target) {
  return createReactiveObject(target, readonlyMap, readonlyHandlers);
}

export function shallowReadonly(target) {
  return createReactiveObject(
    target,
    shallowReadonlyMap,
    shallowReadonlyHandlers
  );
}

export function isProxy(value) {
  return isReactive(value) || isReadonly(value);
}

export function isReadonly(value) {
  return !!value[ReactiveFlags.IS_READONLY];
}

export function isReactive(value) {
  // 如果 value 是 proxy 的话
  // 会触发 get 操作，而在 createGetter 里面会判断
```

```
  // 如果 value 是普通对象的话
  // 那么会返回 undefined ，那么就需要转换成布尔值
  return !!value[ReactiveFlags.IS_REACTIVE];
}

export function toRaw(value) {
  // 如果 value 是 proxy 的话 ,那么直接返回就可以了
  // 因为会触发 createGetter 内的逻辑
  // 如果 value 是普通对象的话，
  // 我们就应该返回普通对象
  // 只要不是 proxy ，只要是得到了 undefined 的话，那么就一定是普通对象
  // TODO 这里和源码里面实现的不一样，不确定后面会不会有问题
  if (!value[ReactiveFlags.RAW]) {
    return value;
  }

  return value[ReactiveFlags.RAW];
}

function createReactiveObject(target, proxyMap, baseHandlers) {
  // 核心就是 proxy
  // 目的是可以侦听到用户 get 或者 set 的动作

  // 如果命中的话就直接返回就好了
  // 使用缓存做的优化点
  const existingProxy = proxyMap.get(target);
  if (existingProxy) {
    return existingProxy;
  }

  const proxy = new Proxy(target, baseHandlers);

  // 把创建好的 proxy 给存起来，
  proxyMap.set(target, proxy);
  return proxy;
}
```

- ref.ts

```
import { trackEffects, triggerEffects, isTracking } from "./effect";
import { createDep } from "./dep";
import { isObject, hasChanged } from "@mini-vue/shared";
import { reactive } from "./reactive";

export class RefImpl {
```

```
  private _rawValue: any;
  private _value: any;
  public dep;
  public __v_isRef = true;

  constructor(value) {
    this._rawValue = value;
    // 看看value 是不是一个对象，如果是一个对象的话
    // 那么需要用 reactive 包裹一下
    this._value = convert(value);
    this.dep = createDep();
  }

  get value() {
    // 收集依赖
    trackRefValue(this);
    return this._value;
  }

  set value(newValue) {
    // 当新的值不等于老的值的话，
    // 那么才需要触发依赖
    if (hasChanged(newValue, this._rawValue)) {
      // 更新值
      this._value = convert(newValue);
      this._rawValue = newValue;
      // 触发依赖
      triggerRefValue(this);
    }
  }
}

export function ref(value) {
  return createRef(value);
}

function convert(value) {
  return isObject(value) ? reactive(value) : value;
}

function createRef(value) {
  const refImpl = new RefImpl(value);

  return refImpl;
}

export function triggerRefValue(ref) {
```

```
54    triggerEffects(ref.dep);
55  }
56
57  export function trackRefValue(ref) {
58    if (isTracking()) {
59      trackEffects(ref.dep);
60    }
61  }
62
63  // 这个函数的目的是
64  // 帮助解构 ref
65  // 比如在 template 中使用 ref 的时候，直接使用就可以了
66  // 例如： const count = ref(0) -> 在 template 中使用的话 可以直接 count
67  // 解决方案就是通过 proxy 来对 ref 做处理
68
69  const shallowUnwrapHandlers = {
70    get(target, key, receiver) {
71      // 如果里面是一个 ref 类型的话，那么就返回 .value
72      // 如果不是的话，那么直接返回value 就可以了
73      return unRef(Reflect.get(target, key, receiver));
74    },
75    set(target, key, value, receiver) {
76      const oldValue = target[key];
77      if (isRef(oldValue) && !isRef(value)) {
78        return (target[key].value = value);
79      } else {
80        return Reflect.set(target, key, value, receiver);
81      }
82    },
83  };
84
85  // 这里没有处理 objectWithRefs 是 reactive 类型的时候
86  // TODO reactive 里面如果有 ref 类型的 key 的话， 那么也是不需要调用 ref.value 的
87  // （but 这个逻辑在 reactive 里面没有实现）
88  export function proxyRefs(objectWithRefs) {
89    return new Proxy(objectWithRefs, shallowUnwrapHandlers);
90  }
91
92  // 把 ref 里面的值拿到
93  export function unRef(ref) {
94    return isRef(ref) ? ref.value : ref;
95  }
96
97  export function isRef(value) {
98    return !!value.__v_isRef;
99  }
```

- effect

```ts
export function effect(fn, options = {}) {
  const _effect = new ReactiveEffect(fn);

  // 把用户传过来的值合并到 _effect 对象上去
  // 缺点就是不是显式的，看代码的时候并不知道有什么值
  extend(_effect, options);
  _effect.run();

  // 把 _effect.run 这个方法返回
  // 让用户可以自行选择调用的时机（调用 fn）
  const runner: any = _effect.run.bind(_effect);
  runner.effect = _effect;
  return runner;
}

export function stop(runner) {
  runner.effect.stop();
}
```

- computed

```ts
import { createDep } from "./dep";
import { ReactiveEffect } from "./effect";
import { trackRefValue, triggerRefValue } from "./ref";

export class ComputedRefImpl {
  public dep: any;
  public effect: ReactiveEffect;

  private _dirty: boolean;
  private _value

  constructor(getter) {
    this._dirty = true;
    this.dep = createDep();
    this.effect = new ReactiveEffect(getter, () => {
      // scheduler
      // 只要触发了这个函数说明响应式对象的值发生改变了
      // 那么就解锁，后续在调用 get 的时候就会重新执行，所以会得到最新的值
      if (this._dirty) return;

      this._dirty = true;
```

```
22        triggerRefValue(this);
23      });
24    }
25
26    get value() {
27      // 收集依赖
28      trackRefValue(this);
29      // 锁上，只可以调用一次
30      // 当数据改变的时候才会解锁
31      // 这里就是缓存实现的核心
32      // 解锁是在 scheduler 里面做的
33      if (this._dirty) {
34        this._dirty = false;
35        // 这里执行 run 的话，就是执行用户传入的 fn
36        this._value = this.effect.run();
37      }
38
39      return this._value;
40    }
41  }
42
43  export function computed(getter) {
44    return new ComputedRefImpl(getter);
45  }
```

5. 工程化：如果要在esbuild中支持ES5打包，如何做？

> github：https://github.com/encode-studio-fe/encode-bundle

ES5 is not supported well

Transforming ES6+ syntax to ES5 is not supported yet. However, if you're using esbuild to transform ES5 code, you should still set the target to `es5`. This prevents esbuild from introducing ES6 syntax into your ES5 code. For example, without this flag the object literal `{x: x}` will become `{x}` and the string `"a\nb"` will become a multi-line template literal when minifying. Both of these substitutions are done because the resulting code is shorter, but the substitutions will not be performed if the target is `es5`.

可以发现，官网是不支持将ESNext的写法转成ES5的写法，那我们就不能使用esbuild compiler 至低版本吗？

此题的考察目的在于对构建工具的理解，这也是前端高级工程师在日常环境下需要解决的问题：

1. 平时开发过程中，有没有对构建工具的具体使用；
2. 有没有尝试通过构建工具自带的扩展插件集成至特殊的业务场景；

此题可以通过esbuild的插件实现：

1. 将代码编译成ES2020；
2. 使用SWC编译成ES5；

```
export const es5 = (): Plugin => {
  let enabled = false;
  return {
    name: 'es5-target',

    esbuildOptions(options) {
      if (options.target === 'es5') {
        options.target = 'es2020';
        enabled = true;
      }
    },

    async renderChunk(code, info) {
      if (!enabled || !/\.(cjs|js)$/.test(info.path)) {
        return;
      }
      const swc: typeof import('@swc/core') = localRequire('@swc/core');

      if (!swc) {
        throw new PrettyError(
          '@swc/core is required for es5 target. Please install it with `npm ins
        );
      }

      const result = await swc.transform(code, {
        filename: info.path,
        sourceMaps: this.options.sourcemap,
        minify: Boolean(this.options.minify),
        jsc: {
          target: 'es5',
          parser: {
            syntax: 'ecmascript',
          },
          minify:
            this.options.minify === true
              ? {
                  compress: false,
                  mangle: {
                    reserved: this.options.globalName ? [this.options.globalName
                  },
                }
              : undefined,
        },
```

```
      });
      return {
        code: result.code,
        map: result.map,
      };
    },
  };
};
```

6. 面试真题汇总

6.1 JS

1. 实现 FIFO、LRU、LFU 算法；
2. 手写 Promise ；
3. JS 的垃圾回收机制是什么；
4. 使用 JS 实现带并发的一部任务调度器；
5. 什么是编译原理；

6.2 Vue

1. 介绍 Vue2 和 Vue3 diff 的区别；
2. 介绍 Vue2 optional API 和 Vue3 Composition API 的区别；
3. 介绍 Vue2 和 Vue3 响应式的区别；
4. 如何实现 shallowReactive 和 ShallowReadonly；
5. Create-vite 实现了哪些功能；

6.3 React

1. HOC 与 Hooks 的区别?
2. React fiber 是什么?
3. 为什么 React 升级后 componentWillXXX 都成为了UNSAFE?
4. Create-React-App 实现了哪些功能?

6.4 工程化

1. 使用的包管理工具是什么？常见的包管理工具有哪些？各自有哪些特点？
2. 了解哪些前端构建工具？主要解决了哪些事情？

3. 有使用过微前端吗？解决了哪些问题？
4. 什么是 bundleless？实现 bundleless 的前提是什么？
5. multipleRepo 和 monorepo 的区别是什么？有使用过哪些 monorepo？
6. 使用过哪些自动化构建过程？使用过 jinkens 或者 docker 吗？
7. 使用 babel 实现 JS 的可选链；
8. 有使用过 babel 的插件吗？babel-plugin-import 实现了哪些功能？是如何实现的？
9. 能否借助现代构建功能，手动先将ESNext打包至ES5的功能？
10. 实现一个 CLI 应该包含哪些功能？

6.5 多端构建

1. 为什么Taro 或者 uniapp 能够实现将 React 或者 Vue 的代码最终生成小程序代码？
2. 小程序多端框架的 compiler 和 runtime 方式分别是如何实现的？
3. React Native、Weex 和 Flutter 的区别？

6.6 性能优化

1. 如何实现 PWA？
2. 如何实现无限滚动？
3. 如何监控前端数据异常？