2021上半年各大厂核心面试题解析（第三期）

react fiber?

讲react fiber之前, 咱们先来看一下普通的函数如何执行?

咱们用while来示例, 可以看出一旦开始, 直到task清空, 期间的行为咱们完全无法控制.

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

而如果我们用generator来写, 其实就能在函数执行时通过yeild中断, 通过next去恢复.

```
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)
  }
}
```

而react fiber的核心目的就是为了使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。

这样的话高性能要求的一些dom计算在设备上就不会显得很卡顿, 而是会一帧一帧的有规律的执行, 看起来就十分流畅.

那么有几个问题.

1. generator有类似的功能, 为什么不直接使用?

React开发人员在git issue里回答过这个问题. 总结起来主要的就是两点:

- 要使用generator的话, 需要将涉及的所有代码都包装成generator * 的形式, 非常麻烦
- generator内部是有状态的, 很难在恢复执行的时候获取之前的状态.

```
function* doWork(a, b, c) {
  var x = doExpensiveWorkA(a);
  yield;
  var y = x + doExpensiveWorkB(b);
  yield;
  var z = y + doExpensiveWorkC(c);
  return z;
}
```

比如这段代码, 如果想在多个时间分片内执行, 而当我们在之前的时间片内已经执行完了doExpensiveWorkA 和 doExpensiveWorkB, 还没执行doExpensiveWorkC, 但是此时b被更新了. 那么在新的时间分片里, 我们只能沿用之前获取到的x和y的结果, 来执行doExpensiveWorkC. 而我们无法获取到更新后的b的值, 再来继续做doExpensiveWorkC的计算.

2. 怎么判定现在有更高优先级的任务?

而我们真正的代码中其实无法真正的判断是否有更高优先级的任务, 只能来约定一个合理的执行时间, 当过了这个执行时间后任务仍没有执行完成的话, 就中断当前任务. 并且将控制权交还给浏览器.

而正常情况下, 我们一般是按照每秒60帧, 也就是每帧16ms的刷新是人眼能感知的最低限度.

然而浏览器恰好提供了这样的方法, requestIdleCallback。

https://developer.mozilla.org/zh-CN/docs/Web/API/Window/requestIdleCallback

requestIdleCallback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参数，表示浏览器有多少时间供我们执行。

那么说了半天, 都是要在浏览器有空的时候执行, 那浏览器什么时候才有空呢?

- 浏览器在一帧内都要做什么事情

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

而只有在做完这些本职工作后, 剩下的时间才是有空的时间, 也就是执行我们 requestIdleCallback 回调的时间.

- 浏览器很忙怎么办?

浏览器可能一直很忙, 忙到几十帧都没空去执行requestIdleCallback, 那咱们的回调岂不是永远无法执行了?

放心, 浏览器给我们提供了一个timeout参数, 当超过这个超时时间并且回调还没执行时, 在下一帧这个回调会被强制执行.

- 兼容性?

众所周知requestIdleCallback的兼容性是很差的, react通过messageChannel模拟实现了 requestIdleCallback 的功能.

- timeout超时后就一定要被执行吗?

在react里不是的, react预订了5个优先级, 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.

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

什么是高阶组件？高阶组件能用来做什么？

高阶组件简称 HOC, 即为 High Order Components.

```
A higher-order component is a function that takes a component and returns a new component.
```

咱们稍微分解一下官方定义，可以得到以下信息：

1. 高阶组件是一个**函数**
2. 入参：原 react 组件
3. 出参：新 react 组件
4. 高阶组件是一个纯函数, 它不应该有任何副作用, 比如修改传入的 react 组件(当然这个不是上面那句话能看出来的，这一点称之为约束或者规范更合理一些)

所以，高阶组件是一个函数，接收一个组件，返回一个组件。

先来写一个高阶函数

在写高阶组件之前, 咱们根据上面的 4 条信息, 先写一个简单的高阶函数的实现试一下。

比如现在有两个函数，

```
function helloWorld() {
    const myName = sessionStorage.getItem("lubai");
    console.log("hello, beautiful world !! my name is " + myName);
}

function byeWorld() {
    const myName = sessionStorage.getItem("lubai");
    console.log("bye, ugly world !! my name is " + myName);
}

helloWorld();
byeWorld();
```

两个函数一个表达了对世界的渴望与好奇, 是一种新生；一个表达了对世界的失望与无奈，是一种死去；文艺的一匹
但是我们可以发现, myName 的获取逻辑都是一样的, 而我们重复写了两遍, 只有 console.log 的逻辑是不同的。
万一以后 myName 的获取逻辑变了怎么办？？ 我们能不能封装一下?

所以我们可以写一个中间函数, 里面包含获取 myName 的逻辑。

```
function helloWorld(myName) {
    console.log("hello, beautiful world !! my name is " + myName);
}

function byeWorld(myName) {
    console.log("bye, ugly world !! my name is " + myName);
}

function wrapWithUserName(wrappedFunc) {
    const tempFunction = () => {
        const myName = sessionStorage.getItem("qiuku");
        wrappedFunc(myName);
    };
    return tempFunction;
}

wrapWithUserName(helloWorld)();
wrapWithUserName(byeWorld)();
```

怎样写一个高阶组件

平时看到的大概是这样的

```
export const NewComponent = hoc(WrappedComponent);
```

1. 普通方式

```
接下来咱们用普通方式写一个类的高阶组件
```

2. 装饰器

> 接下来咱们用装饰器方式写一个类的高阶组件

3. 多个高阶组件组合

> 会发现用普通方式书写的话，逻辑会显得非常乱，所以建议使用装饰器的写法。

高阶组件能用来做什么

1. 属性代理

```
1.1 操作 props 其实上面的这几个例子，就是在操作 props
1.2 操作组件实例
```

2. 继承/劫持

什么是 react hooks? React hooks有什么优势?

Hook 即为"钩子", 是 react 16.8 的新特性, 你可以在不写 class 的情况下使用 state 和其他的 react 特性。

凡是 use 开头的 React API 都是 Hooks.

那么为什么要不写 class 呢? hook 相对于 class 又有什么优势呢?

react hooks 有什么优势

先来看一下 class 写组件有什么不足之处吧!

1. 组件间的状态逻辑很难复用

组件间如果有 state 的逻辑是类似的话, class 模式下基本都是用高阶组件来解决的,。

虽然能够解决问题, 但是你会发现, 我们可能需要在组件外部再包一层元素, 会导致层级非常冗余

2. 复杂业务的有状态组件会越来越复杂

比如类组件中都是通过更改 this.state 来达到状态修改的目的的, 但是组件内部太多对 state 的访问和修改, 很难在后期给拆成更细粒度的组件, 就会导致组件越来越庞大。
还有比如设置监听, 比如添加定时器, 我们需要在两个生命周期里完成注册和销毁, 很有可能漏写导致内存问题

```
componentDidMount() {
    const timer = setInterval(() => {});
    this.setState({timer})
}

componentWillUnmount() {
    if (this.state.timer) {
        clearInterval(this.state.timer);
    }
}
```

3. This 指向问题

React 里绑定事件函数有以下四种方法，如果新玩家刚接触，稍不注意就会写错，导致性能上的大大损耗。

```
class App extends React.Component<any, any> {
    handleClick2;

    constructor(props) {
        super(props);
        this.state = {
            num: 1,
            title: " react study",
        };
        this.handleClick2 = this.handleClick1.bind(this);
```

```
    }

    handleClick1() {
        this.setState({
            num: this.state.num + 1,
        });
    }

    handleClick3 = () => {
        this.setState({
            num: this.state.num + 1,
        });
    };

    render() {
        return (
            <div>
                <h2>Ann, {this.state.num}</h2>
                {/* 在render函数里绑定this，由于bind会返回一个新函数，所以每次父组件刷新都会导致子组件的重新刷新，就算子组件的其他props没有改变。*/}
                <button onClick={this.handleClick1.bind(this)}>btn1</button>
                {/* 构造函数内绑定this，每次父组件刷新的时候，如果传递给子组件的其他props不变，子组件就不会刷新。*/}
                <button onClick={this.handleClick2}>btn2</button>
                {/* 使用箭头函数，每次都会生成一个新的箭头函数，每次父组件刷新的时候，如果传递给子组件的其他props不变，子组件就不会刷新*/}
                <button onClick={() => this.handleClick1()}>btn3</button>
                {/* 使用类里定义的箭头函数，和handleClick2原理一样，但是比第二种更简洁*/}
                <button onClick={this.handleClick3}>btn4</button>
            </div>
        );
    }
}
```

hooks的优点

而 hooks 的优点, 一对比就比较明显了

1. 能优化类组件的三大问题
2. 能在无需修改组件结构的情况下复用状态逻辑（自定义 Hooks ）
3. 能将组件中相互关联的部分拆分成更小的函数（比如设置订阅或请求数据）
4. 副作用的概念

副作用指那些没有发生在数据向视图转换过程中的逻辑，如 ajax 请求、访问原生 dom 元素、本地持久化缓存、绑定/解绑事件、添加订阅、设置定时器、记录日志等。
以往这些副作用都是写在类组件生命周期函数中的。
而 useEffect 在全部渲染完毕后才会执行，useLayoutEffect 会在浏览器 layout 之后，painting 之前执行。
而且比如绑定/解绑事件都可以写在一个副作用函数里了, 不会再散落在各地难以维护。

react hooks 的注意事项

1. 只能在函数内部的最外层调用 Hook，不要在循环、条件判断或者子函数中调用
2. 只能在 React 的函数组件中调用 Hook，不要在其他 JavaScript 函数中调用

react hooks 是怎么实现的

说了这么多, 优点啊, 注意事项啊, 大家可能比较懵逼。

1. 为什么不能在循环或者判断条件中使用？？
2. 为什么 useEffect 的第二个参数是空数组, 就相当于 componentDidMount 只执行一次？？
3. 自定义 hook 怎么操作组件的?

接下来我们来实现一下简单的 Hooks, 一看就懂了。

1. useState

先来看一下 useState 是怎么使用的

```
const [count, setCount] = useState(0);
```

传入一个初始值, 返回一个状态值和一个设置状态的方法。

咱们先来实现一个简易的 useState。 // 代码 Mock-UseState1-Counter.tsx

可以看到这个 useState 只支持设置一次的, 如果是多个 useState 的话就无法满足需求了。

2. 多个useState

那么咱们来看看怎么支持多个 useState！！

前面 useState 的简单实现里，初始的状态是保存在一个全局变量中的。
以此类推，多个状态，应该是保存在一个专门的全局容器中。这个容器，就是一个朴实无华的 Array 对象。具体过程如下：

- 第一次渲染时候，根据 useState 顺序，逐个声明 state 并且将其放入全局 Array 中。
- 每次声明 state，都要将 cursor 增加 1。
- 更新 state，触发再次渲染的时候，cursor 被重置为 0。
- 按照 useState 的声明顺序，依次拿出最新的 state 的值，视图更新。

那么明白了原理之后, 你可能也会想明白为什么不能在循环或者判断条件中使用了吧？？

因为是按照数组索引入存储的, 如果在判断中使用, 某个 state 没有按照对应索引存储，那么后面的 state 索引也会有问题。

// 代码 Mock-UseState2-Counter.tsx

3. useEffect

来看一下最基本的用法

```
useEffect(() => {
    console.log(count);
}, [count]);
```

它有如下四个特点

- 有两个参数 callback 和 dependencies 数组
- 如果 dependencies 不存在，那么 callback 每次 render 都会执行
- 如果 dependencies 存在，只有当它发生了变化， callback 才会执行
- 如果 dependencies 为空数组, 则只执行一次 callback

接下来来实现一个简易 useEffect 吧！！

// 代码 Mock-UseEffect1-Counter.tsx

它同样有个问题, 就是只能注册一个 useEffect，利用相同的 cursor+array 的思想，实现多个 useEffect 的注册吧

// 代码 Mock-UseEffect2-Counter.tsx

手写代码

1. promise-allsettled.js
2. promise-limit.js