前端关于面试你可能需要收集的面试题

Proxy 可以实现什么功能?

在 Vue3.0 中通过 `Proxy` 来替换原本的 `Object.defineProperty` 来实现数据响应式。

Proxy 是 ES6 中新增的功能，它可以用来自定义对象中的操作。

javascript 复制代码

```
let p = new Proxy(target, handler)
```

`target` 代表需要添加代理的对象， `handler` 用来自定义对象中的操作，比如可以用来自定义 `set` 或者 `get` 函数。

下面来通过 `Proxy` 来实现一个数据响应式：

javascript 复制代码

```
let onWatch = (obj, setBind, getLogger) => {
  let handler = {
    get(target, property, receiver) {
      getLogger(target, property)
      return Reflect.get(target, property, receiver)
    },
    set(target, property, value, receiver) {
      setBind(value, property)
      return Reflect.set(target, property, value)
    }
  }
  return new Proxy(obj, handler)
}
let obj = { a: 1 }
let p = onWatch(
  obj,
  (v, property) => {
    console.log(`监听到属性${property}改变为${v}`)
  },
  (target, property) => {
    console.log(`'${property}' = ${target[property]}`)
  }
)
p.a = 2 // 监听到属性a改变
```

```
p.a // 'a' = 2
```

在上述代码中，通过自定义 `set` 和 `get` 函数的方式，在原本的逻辑中插入了我们的函数逻辑，实现了在对对象任何属性进行读写时发出通知。

当然这是简单版的响应式实现，如果需要实现一个 Vue 中的响应式，需要在 `get` 中收集依赖，在 `set` 派发更新，之所以 Vue3.0 要使用 `Proxy` 替换原本的 API 原因在于 `Proxy` 无需一层层递归为每个属性添加代理，一次即可完成以上操作，性能上更好，并且原本的实现有一些数据更新不能监听到，但是 `Proxy` 可以完美监听到任何方式的数据改变，唯一缺陷就是浏览器的兼容性不好。

script标签中defer和async的区别

如果没有defer或async属性，浏览器会立即加载并执行相应的脚本。它不会等待后续加载的文档元素，读取到就会开始加载和执行，这样就阻塞了后续文档的加载。

defer 和 async属性都是去异步加载外部的JS脚本文件，它们都不会阻塞页面的解析，其区别如下：

- **执行顺序：** 多个带async属性的标签，不能保证加载的顺序；多个带defer属性的标签，按照加载顺序执行；
- **脚本是否并行执行：async属性，表示后续文档的加载和执行与js脚本的加载和执行是并行进行的**，即异步执行；defer属性，加载后续文档的过程和js脚本的加载(此时仅加载不执行)是并行进行的(异步)，js脚本需要等到文档所有元素解析完成之后才执行，DOMContentLoaded事件触发执行之前。

Promise.all

描述：所有 `promise` 的状态都变成 `fulfilled` ，就会返回一个状态为 `fulfilled` 的数组(所有 `promise` 的 `value`)。只要有一个失败，就返回第一个状态为 `rejected` 的 `promise` 实例的 `reason` 。

实现：

javascript 复制代码

```
Promise.all = function(promises) {
    return new Promise((resolve, reject) => {
        if(Array.isArray(promises)) {
```

```
            if(promises.length === 0) return resolve(promises);
            let result = [];
            let count = 0;
            promises.forEach((item, index) => {
                Promise.resolve(item).then(
                    value => {
                        count++;
                        result[index] = value;
                        if(count === promises.length) resolve(result);
                    },
                    reason => reject(reason)
                );
            })
        }
        else return reject(new TypeError("Argument is not iterable"));
    });
}
```

ES6新特性

javascript 复制代码

1.**ES6**引入来严格模式
    变量必须声明后在使用
    函数的参数不能有同名属性，否则报错
    不能使用**with**语句 (说实话我基本没用过)
    不能对只读属性赋值，否则报错
    不能使用前缀0表示八进制数,否则报错 (说实话我基本没用过)
    不能删除不可删除的数据，否则报错
    不能删除变量**delete** prop，会报错，只能删除属性**delete** global[prop]
    eval不会在它的外层作用域引入变量
    eval和arguments不能被重新赋值
    arguments不会自动反映函数参数的变化
    不能使用arguments.caller (说实话我基本没用过)
    不能使用arguments.callee (说实话我基本没用过)
    禁止this指向全局对象
    不能使用fn.caller和fn.arguments获取函数调用的堆栈 (说实话我基本没用过)
    增加了保留字（比如protected、**static**和interface）

2.关于**let**和**const**新增的变量声明

3.变量的解构赋值

4.字符串的扩展
    includes()：返回布尔值，表示是否找到了参数字符串。
    startsWith()：返回布尔值，表示参数字符串是否在原字符串的头部。
    endsWith()：返回布尔值，表示参数字符串是否在原字符串的尾部。
5.数值的扩展

Number.isFinite()用来检查一个数值是否为有限的（finite）。

Number.isNaN()用来检查一个值是否为NaN。

6.函数的扩展

函数参数指定默认值

7.数组的扩展

扩展运算符

8.对象的扩展

对象的解构

9.新增symbol数据类型

10.Set 和 Map 数据结构

ES6 提供了新的数据结构 Set。它类似于数组，但是成员的值都是唯一的，没有重复的值。 Set 本身是一个构造函数，

Map它类似于对象，也是键值对的集合，但是“键”的范围不限于字符串，各种类型的值（包括对象）都可以当作键。

11.Proxy

Proxy 可以理解成，在目标对象之前架设一层“拦截”，外界对该对象的访问

都必须先通过这层拦截，因此提供了一种机制，可以对外界的访问进行过滤和改写。

Proxy 这个词的原意是代理，用在这里表示由它来“代理”某些操作，可以译为“代理器”。

Vue3.0使用了proxy

12.Promise

Promise 是异步编程的一种解决方案，比传统的解决方案—回调函数和事件—更合理和更强大。

特点是：

对象的状态不受外界影响。

一旦状态改变，就不会再变，任何时候都可以得到这个结果。

13.async 函数

async函数对 Generator 函数的区别：

（1）内置执行器。

Generator 函数的执行必须靠执行器，而async函数自带执行器。也就是说，async函数的执行，与普通函数一模一样，

（2）更好的语义。

async和await，比起星号和yield，语义更清楚了。async表示函数里有异步操作，await表示紧跟在后面的表达式需要

（3）正常情况下，await命令后面是一个 Promise 对象。如果不是，会被转成一个立即resolve的 Promise 对象。

（4）返回值是 Promise。

async函数的返回值是 Promise 对象，这比 Generator 函数的返回值是 Iterator 对象方便多了。你可以用then方

14.Class

class跟let、const一样：不存在变量提升、不能重复声明...

ES6 的class可以看作只是一个语法糖，它的绝大部分功能

ES5 都可以做到，新的class写法只是让对象原型的写法更加清晰、更像面向对象编程的语法而已。

15.Module

ES6 的模块自动采用严格模式，不管你有没有在模块头部加上"use strict";。

import和export命令以及export和export default的区别

偏函数

什么是偏函数？偏函数就是将一个 n 参的函数转换成固定 x 参的函数，剩余参数（n - x）将在下次调用全部传入。举个例子：

```javascript 复制代码
function add(a, b, c) {
    return a + b + c
}
let partialAdd = partial(add, 1)
partialAdd(2, 3)
```

发现没有，其实偏函数和函数柯里化有点像，所以根据函数柯里化的实现，能够能很快写出偏函数的实现：

```javascript 复制代码
function partial(fn, ...args) {
    return (...arg) => {
        return fn(...args, ...arg)
    }
}
```

如上这个功能比较简单，现在我们希望偏函数能和柯里化一样能实现占位功能，比如：

```javascript 复制代码
function clg(a, b, c) {
    console.log(a, b, c)
}
let partialClg = partial(clg, '_', 2)
partialClg(1, 3)  // 依次打印：1, 2, 3
```

_ 占的位其实就是 1 的位置。相当于：partial(clg, 1, 2)，然后 partialClg(3)。明白了原理，我们就来写实现：

```javascript 复制代码
function partial(fn, ...args) {
    return (...arg) => {
        args[index] =
        return fn(...args, ...arg)
    }
}
```

数组扁平化

ES5 递归写法 —— isArray()、concat()

javascript 复制代码

```
function flat11(arr) {
    var res = [];
    for (var i = 0; i < arr.length; i++) {
        if (Array.isArray(arr[i])) {
            res = res.concat(flat11(arr[i]));
        } else {
            res.push(arr[i]);
        }
    }
    return res;
}
```

如果想实现第二个参数（指定“拉平”的层数），可以这样实现，后面的几种可以自己类似实现：

javascript 复制代码

```
function flat(arr, level = 1) {
    var res = [];
    for(var i = 0; i < arr.length; i++) {
        if(Array.isArray(arr[i]) || level >= 1) {
            res = res.concat(flat(arr[i]), level - 1);
        }
        else {
            res.push(arr[i]);
        }
    }
    return res;
}
```

ES6 递归写法 — reduce()、concat()、isArray()

javascript 复制代码

```
function flat(arr) {
    return arr.reduce(
        (pre, cur) => pre.concat(Array.isArray(cur) ? flat(cur) : cur), []
    );
}
```

ES6 迭代写法 — 扩展运算符(...)、some()、concat()、isArray()

ES6 的扩展运算符(...) 只能扁平化一层

```javascript 复制代码
function flat(arr) {
    return [].concat(...arr);
}
```

全部扁平化：遍历原数组，若 `arr` 中含有数组则使用一次扩展运算符，直至没有为止。

```javascript 复制代码
function flat(arr) {
    while(arr.some(item => Array.isArray(item))) {
        arr = [].concat(...arr);
    }
    return arr;
}
```

toString/join & split

调用数组的 `toString()/join()` 方法（它会自动扁平化处理），将数组变为字符串然后再用 `split` 分割还原为数组。由于 `split` 分割后形成的数组的每一项值为字符串，所以需要用一个 `map` 方法遍历数组将其每一项转换为数值型。

```javascript 复制代码
function flat(arr){
    return arr.toString().split(',').map(item => Number(item));
    // return arr.join().split(',').map(item => Number(item));
}
```

使用正则

`JSON.stringify(arr).replace(/[|]/g, '')` 会先将数组 `arr` 序列化为字符串，然后使用 `replace()` 方法将字符串中所有的 `[` 或 `]` 替换成空字符，从而达到扁平化处理，此时的结果为 `arr` 不包含 `[]` 的字符串。最后通过 `JSON.parse()` 解析字符串。

```javascript 复制代码
function flat(arr) {
    return JSON.parse("[" + JSON.stringify(arr).replace(/\[|\]/g,'') + "]");
}
```

类数组转化为数组

类数组是具有 `length` 属性，但不具有数组原型上的方法。常见的类数组有 `arguments` 、DOM 操作方法返回的结果(如 `document.querySelectorAll('div')`)等。

扩展运算符(...)

注意：扩展运算符只能作用于 `iterable` 对象，即拥有 `Symbol(Symbol.iterator)` 属性值。

javascript 复制代码

```
let arr = [...arrayLike]
```

Array.from()

javascript 复制代码

```
let arr = Array.from(arrayLike);
```

Array.prototype.slice.call()

javascript 复制代码

```
let arr = Array.prototype.slice.call(arrayLike);
```

Array.apply()

javascript 复制代码

```
let arr = Array.apply(null, arrayLike);
```

concat + apply

javascript 复制代码

```
let arr = Array.prototype.concat.apply([], arrayLike);
```

参考 前端进阶面试题详细解答

代码输出结果

javascript 复制代码

```javascript
console.log('1');

setTimeout(function() {
    console.log('2');
    process.nextTick(function() {
        console.log('3');
    })
    new Promise(function(resolve) {
        console.log('4');
        resolve();
    }).then(function() {
        console.log('5')
    })
})
process.nextTick(function() {
    console.log('6');
})
new Promise(function(resolve) {
    console.log('7');
    resolve();
}).then(function() {
    console.log('8')
})

setTimeout(function() {
    console.log('9');
    process.nextTick(function() {
        console.log('10');
    })
    new Promise(function(resolve) {
        console.log('11');
        resolve();
    }).then(function() {
        console.log('12')
    })
})
```

输出结果如下：

javascript 复制代码

```
1
7
6
8
```

```
2
4
3
5
9
11
10
12
```

（1）第一轮事件循环流程分析如下：

- 整体script作为第一个宏任务进入主线程，遇到 `console.log` ，输出1。
- 遇到 `setTimeout` ，其回调函数被分发到宏任务Event Queue中。暂且记为 `setTimeout1` 。
- 遇到 `process.nextTick()` ，其回调函数被分发到微任务Event Queue中。记为 `process1` 。
- 遇到 `Promise` ， `new Promise` 直接执行，输出7。 `then` 被分发到微任务Event Queue中。记为 `then1` 。
- 又遇到了 `setTimeout` ，其回调函数被分发到宏任务Event Queue中，记为 `setTimeout2` 。

宏任务Event Queue	微任务Event Queue
setTimeout1	process1
setTimeout2	then1

上表是第一轮事件循环宏任务结束时各Event Queue的情况，此时已经输出了1和7。发现了 `process1` 和 `then1` 两个微任务：

- 执行 `process1` ，输出6。
- 执行 `then1` ，输出8。

第一轮事件循环正式结束，这一轮的结果是输出1，7，6，8。

（2）第二轮时间循环从 `**setTimeout1**` 宏任务开始：

- 首先输出2。接下来遇到了 `process.nextTick()` ，同样将其分发到微任务Event Queue中，记为 `process2` 。
- `new Promise` 立即执行输出4， `then` 也分发到微任务Event Queue中，记为 `then2` 。

宏任务Event Queue	微任务Event Queue
setTimeout2	process2
	then2

第二轮事件循环宏任务结束，发现有 `process2` 和 `then2` 两个微任务可以执行：

- 输出3。
- 输出5。

第二轮事件循环结束，第二轮输出2，4，3，5。

（3）第三轮事件循环开始，此时只剩setTimeout2了，执行。

- 直接输出9。
- 将 `process.nextTick()` 分发到微任务Event Queue中。记为 `process3` 。
- 直接执行 `new Promise` ，输出11。
- 将 `then` 分发到微任务Event Queue中，记为 `then3` 。

宏任务Event Queue	微任务Event Queue
	process3
	then3

第三轮事件循环宏任务执行结束，执行两个微任务 `process3` 和 `then3` ：

- 输出10。
- 输出12。

第三轮事件循环结束，第三轮输出9，11，10，12。

整段代码，共进行了三次事件循环，完整的输出为1，7，6，8，2，4，3，5，9，11，10，12。

说一下原型链和原型链的继承吧

- 所有普通的 [[Prototype]] 链最终都会指向内置的 Object.prototype，其包含了 JavaScript 中许多通用的功能
- 为什么能创建 “类”，借助一种特殊的属性：所有的函数默认都会拥有一个名为 prototype 的共有且不可枚举的属性，它会指向另外一个对象，这个对象通常被称为函数的原型

javascript 复制代码

```javascript
function Person(name) {
  this.name = name;
}

Person.prototype.constructor = Person
```

- 在发生 new 构造函数调用时，会将创建的新对象的 [[Prototype]] 链接到 Person.prototype 指向的对象，这个机制就被称为原型链继承
- 方法定义在原型上，属性定义在构造函数上
- 首先要说一下 JS 原型和实例的关系：每个构造函数 （constructor）都有一个原型对象（prototype），这个原型对象包含一个指向此构造函数的指针属性，通过 new 进行构造函数调用生成的实例，此实例包含一个指向原型对象的指针，也就是通过 [[Prototype]] 链接到了这个原型对象
- 然后说一下 JS 中属性的查找：当我们试图引用实例对象的某个属性时，是按照这样的方式去查找的，首先查找实例对象上是否有这个属性，如果没有找到，就去构造这个实例对象的构造函数的 prototype 所指向的对象上去查找，如果还找不到，就从这个 prototype 对象所指向的构造函数的 prototype 原型对象上去查找
- 什么是原型链：这样逐级查找形似一个链条，且通过 [[Prototype]] 属性链接，所以被称为原型链
- 什么是原型链继承，类比类的继承：当有两个构造函数 A 和 B，将一个构造函数 A 的原型对象的，通过其 [[Prototype]] 属性链接到另外一个 B 构造函数的原型对象时，这个过程被称之为原型继承。

** 标准答案更正确的解释**

什么是原型链?

当对象查找一个属性的时候，如果没有在自身找到，那么就会查找自身的原型，如果原型还没有找到，那么会继续查找原型的原型，直到找到 Object.prototype 的原型时，此时原型为

null，查找停止。 这种通过 通过原型链接的逐级向上的查找链被称为原型链

什么是原型继承?

一个对象可以使用另外一个对象的属性或者方法，就称之为继承。具体是通过将这个对象的原型设置为另外一个对象，这样根据原型链的规则，如果查找一个对象属性且在自身不存在时，就会查找另外一个对象，相当于一个对象可以使用另外一个对象的属性和方法了。

手写题：Promise 原理

javascript 复制代码

```
class MyPromise {
  constructor(fn) {
    this.callbacks = [];
    this.state = "PENDING";
    this.value = null;

    fn(this._resolve.bind(this), this._reject.bind(this));
  }

  then(onFulfilled, onRejected) {
    return new MyPromise((resolve, reject) =>
      this._handle({
        onFulfilled: onFulfilled || null,
        onRejected: onRejected || null,
        resolve,
        reject,
      })
    );
  }

  catch(onRejected) {
    return this.then(null, onRejected);
  }

  _handle(callback) {
    if (this.state === "PENDING") {
      this.callbacks.push(callback);

      return;
    }

    let cb =
      this.state === "FULFILLED" ? callback.onFulfilled : callback.onRejected;
    if (!cb) {
      cb = this.state === "FULFILLED" ? callback.resolve : callback.reject;
      cb(this.value);
```

```
      return;
    }

    let ret;

    try {
      ret = cb(this.value);
      cb = this.state === "FULFILLED" ? callback.resolve : callback.reject;
    } catch (error) {
      ret = error;
      cb = callback.reject;
    } finally {
      cb(ret);
    }
  }

  _resolve(value) {
    if (value && (typeof value === "object" || typeof value === "function")) {
      let then = value.then;

      if (typeof then === "function") {
        then.call(value, this._resolve.bind(this), this._reject.bind(this));

        return;
      }
    }

    this.state === "FULFILLED";
    this.value = value;
    this.callbacks.forEach((fn) => this._handle(fn));
  }

  _reject(error) {
    this.state === "REJECTED";
    this.value = error;
    this.callbacks.forEach((fn) => this._handle(fn));
  }
}

const p1 = new Promise(function (resolve, reject) {
  setTimeout(() => reject(new Error("fail")), 3000);
});

const p2 = new Promise(function (resolve, reject) {
  setTimeout(() => resolve(p1), 1000);
});

p2.then((result) => console.log(result)).catch((error) => console.log(error));
```

代码输出结果

javascript 复制代码

```
var myObject = {
    foo: "bar",
    func: function() {
        var self = this;
        console.log(this.foo);
        console.log(self.foo);
        (function() {
            console.log(this.foo);
            console.log(self.foo);
        }());
    }
};
myObject.func();
```

输出结果：bar bar undefined bar

解析：

1. 首先func是由myObject调用的，this指向myObject。又因为var self = this;所以self指向myObject。
2. 这个立即执行匿名函数表达式是由window调用的，this指向window 。立即执行匿名函数的作用域处于myObject.func的作用域中，在这个作用域找不到self变量，沿着作用域链向上查找self变量，找到了指向 myObject对象的self。

setInterval 模拟 setTimeout

描述：使用 `setInterval` 模拟实现 `setTimeout` 的功能。

思路： `setTimeout` 的特性是在指定的时间内只执行一次，我们只要在 `setInterval` 内部执行 `callback` 之后，把定时器关掉即可。

实现：

javascript 复制代码

```javascript
const mySetTimeout = (fn, time) => {
    let timer = null;
    timer = setInterval(() => {
        // 关闭定时器，保证只执行一次fn，也就达到了setTimeout的效果了
        clearInterval(timer);
        fn();
    }, time);
    // 返回用于关闭定时器的方法
    return () => clearInterval(timer);
}

// 测试
const cancel = mySetTimeout(() => {
    console.log(1);
}, 1000);
// 一秒后打印 1
```

为什么0.1+0.2！== 0.3，如何让其相等

在开发过程中遇到类似这样的问题：

javascript 复制代码

```javascript
let n1 = 0.1, n2 = 0.2
console.log(n1 + n2)  // 0.30000000000000004
```

这里得到的不是想要的结果，要想等于0.3，就要把它进行转化：

javascript 复制代码

```javascript
(n1 + n2).toFixed(2) // 注意，toFixed为四舍五入
```

`toFixed(num)` 方法可把 Number 四舍五入为指定小数位数的数字。那为什么会出现这样的结果呢?

计算机是通过二进制的方式存储数据的，所以计算机计算0.1+0.2的时候，实际上是计算的两个数的二进制的和。0.1的二进制是 `0.0001100110011001100...` （1100循环），0.2的二进制是：`0.00110011001100...` （1100循环），这两个数的二进制都是无限循环的数。那JavaScript是如何处理无限循环的二进制小数呢?

一般我们认为数字包括整数和小数，但是在 JavaScript 中只有一种数字类型：Number，它的实现遵循IEEE 754标准，使用64位固定长度来表示，也就是标准的double双精度浮点数。在二进制科学表示法中，双精度浮点数的小数部分最多只能保留52位，再加上前面的1，其实就是保留53位有效数字，剩余的需要舍去，遵从"0舍1入"的原则。

根据这个原则，0.1和0.2的二进制数相加，再转化为十进制数就是： `0.30000000000000004` 。

下面看一下**双精度数是如何保存**的：

- 第一部分（蓝色）：用来存储符号位（sign），用来区分正负数，0表示正数，占用1位
- 第二部分（绿色）：用来存储指数（exponent），占用11位
- 第三部分（红色）：用来存储小数（fraction），占用52位

对于0.1，它的二进制为：

```javascript 复制代码
0.0001100110011001100110011001100110011001100110011001100110011001 10011...
```

转为科学计数法（科学计数法的结果就是浮点数）：

```javascript 复制代码
1.1001100110011001100110011001100110011001100110011001*2^-4
```

可以看出0.1的符号位为0，指数位为-4，小数位为：

```javascript 复制代码
1001100110011001100110011001100110011001100110011001
```

那么问题又来了，**指数位是负数，该如何保存**呢?

IEEE标准规定了一个偏移量，对于指数部分，每次都加这个偏移量进行保存，这样即使指数是负数，那么加上这个偏移量也就是正数了。由于JavaScript的数字是双精度数，这里就以双精度数为例，它的指数部分为11位，能表示的范围就是0~2047，IEEE固定**双精度数的偏移量为1023。**

- 当指数位不全是0也不全是1时(规格化的数值)，IEEE规定，阶码计算公式为 e-Bias。 此时e最小值是1，则1-1023= -1022，e最大值是2046，则2046-1023=1023，可以看到，这种情况下取值范围是 `-1022~1013` 。

- 当指数位全部是0的时候(非规格化的数值)，IEEE规定，阶码的计算公式为1-Bias，即1-1023= -1022。
- 当指数位全部是1的时候(特殊值)，IEEE规定这个浮点数可用来表示3个特殊值，分别是正无穷，负无穷，NaN。 具体的，小数位不为0的时候表示NaN；小数位为0时，当符号位s=0时表示正无穷，s=1时候表示负无穷。

对于上面的0.1的指数位为-4，-4+1023 = 1019 转化为二进制就是： `1111111011` .

所以，0.1表示为：

javascript 复制代码

```
0 1111111011 1001100110011001100110011001100110011001100110011001
```

说了这么多，是时候该最开始的问题了，如何实现0.1+0.2=0.3呢?

对于这个问题，一个直接的解决方法就是设置一个误差范围，通常称为“机器精度”。对JavaScript来说，这个值通常为2-52，在ES6中，提供了 `Number.EPSILON` 属性，而它的值就是2-52，只要判断 `0.1+0.2-0.3` 是否小于 `Number.EPSILON` ，如果小于，就可以判断为0.1+0.2===0.3

javascript 复制代码

```
function numberepsilon(arg1,arg2){
  return Math.abs(arg1 - arg2) < Number.EPSILON;
}

console.log(numberepsilon(0.1 + 0.2, 0.3)); // true
```

代码输出结果

javascript 复制代码

```
function fn1(){
  console.log('fn1')
}
var fn2

fn1()
fn2()

fn2 = function() {
  console.log('fn2')
}
```

```
fn2()
```

输出结果：

javascript 复制代码

```
fn1
Uncaught TypeError: fn2 is not a function
fn2
```

这里也是在考察变量提升，关键在于第一个fn2()，这时fn2仍是一个undefined的变量，所以会报错fn2不是一个函数。

setTimeout 模拟 setInterval

描述：使用 `setTimeout` 模拟实现 `setInterval` 的功能。

实现：

javascript 复制代码

```
const mySetInterval(fn, time) {
    let timer = null;
    const interval = () => {
        timer = setTimeout(() => {
            fn();  // time 时间之后会执行真正的函数fn
            interval();  // 同时再次调用interval本身
        }, time)
    }
    interval();  // 开始执行
    // 返回用于关闭定时器的函数
    return () => clearTimeout(timer);
}

// 测试
const cancel = mySetInterval(() => console.log(1), 400);
setTimeout(() => {
    cancel();
}, 1000);
// 打印两次1
```

代码输出结果

javascript 复制代码

```
function foo() {
  console.log( this.a );
}

function doFoo() {
  foo();
}

var obj = {
  a: 1,
  doFoo: doFoo
};

var a = 2;
obj.doFoo()
```

输出结果：2

在Javascript中，this指向函数执行时的当前对象。在执行foo的时候，执行环境就是doFoo函数，执行环境为全局。所以，foo中的this是指向window的，所以会打印出2。

代码输出结果

javascript 复制代码

```
function runAsync (x) {
  const p = new Promise(r => setTimeout(() => r(x, console.log(x)), 1000))
  return p
}
Promise.race([runAsync(1), runAsync(2), runAsync(3)])
  .then(res => console.log('result: ', res))
  .catch(err => console.log(err))
```

输出结果如下：

javascript 复制代码

```
1
'result: ' 1
2
3
```

then只会捕获第一个成功的方法，其他的函数虽然还会继续执行，但是不是被then捕获了。

单行、多行文本溢出隐藏

- 单行文本溢出

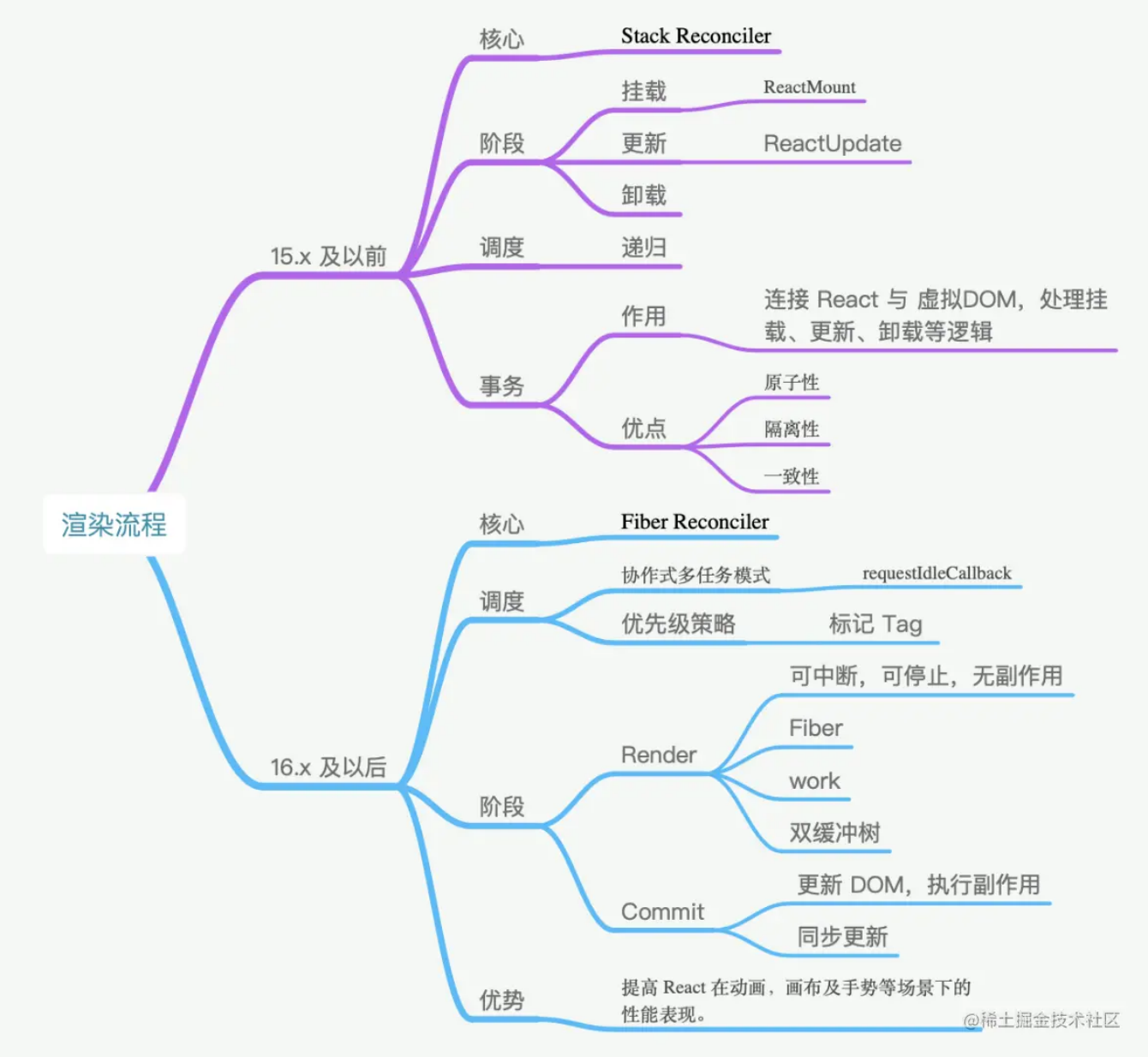
```css 复制代码
overflow: hidden;            // 溢出隐藏
text-overflow: ellipsis;      // 溢出用省略号显示
white-space: nowrap;         // 规定段落中的文本不进行换行
```

- 多行文本溢出

```css 复制代码
overflow: hidden;            // 溢出隐藏
text-overflow: ellipsis;     // 溢出用省略号显示
display:-webkit-box;         // 作为弹性伸缩盒子模型显示。
-webkit-box-orient:vertical; // 设置伸缩盒子的子元素排列方式：从上到下垂直排列
-webkit-line-clamp:3;        // 显示的行数
```

注意：由于上面的三个属性都是 CSS3 的属性，没有浏览器可以兼容，所以要在前面加一个 `-webkit-` 来兼容一部分浏览器。

如何解释 React 的渲染流程



- React 的渲染过程大致一致，但协调并不相同，以 `React 16` 为分界线，分为 `Stack Reconciler` 和 `Fiber Reconciler`。这里的协调从狭义上来讲，特指 React 的 diff 算法，广义上来讲，有时候也指 React 的 `reconciler` 模块，它通常包含了 `diff` 算法和一些公共逻辑。
- 回到 `Stack Reconciler` 中，`Stack Reconciler` 的 `核心调度方式是递归`。`调度的基本处理单位是事务`，它的事务基类是 `Transaction`，这里的 `事务是 React 团队从后端开发中加入的概念`。在 React 16 以前，`挂载主要通过 ReactMount 模块完成`，更新通过 `ReactUpdate` 模块完成，模块之间相互分离，落脚执行点也是事务。
- 在 `React 16` 及以后，协调改为了 `Fiber Reconciler`。它的调度方式主要有两个特点，`第一个是协作式多任务模式`，在这个模式下，线程会定时放弃自己的运行权利，交还给主线程，通过 `requestIdleCallback` 实现。`第二个特点是策略优先级`，调度任务通过标记 `tag` 的方式分优先级执行，比如动画，或者标记为 `high` 的任务可以优先执行。`Fiber Reconciler` 的基本单位是 `Fiber`，`Fiber` 基于过去的 `React Element` 提供了二次封装，提供了指向父、子、兄弟节点的引用，为 `diff` 工作的双链表实现提供了基础。

- 在新的架构下，整个生命周期被划分为 `Render` 和 `Commit` 两个阶段。 `Render` 阶段的执行特点是可中断、可停止、无副作用，主要是通过构造 `workInProgress` 树计算出 `diff`。以 `current` 树为基础，将每个 `Fiber` 作为一个基本单位，自下而上逐个节点检查并构造 workInProgress 树。这个过程不再是递归，而是基于循环来完成
- 在执行上通过 `requestIdleCallback` 来调度执行每组任务，每组中的每个计算任务被称为 `work`，每个 `work` 完成后确认是否有优先级更高的 `work` 需要插入，如果有就让位，没有就继续。优先级通常是标记为动画或者 `high` 的会先处理。每完成一组后，将调度权交回主线程，直到下一次 `requestIdleCallback` 调用，再继续构建 `workInProgress` 树
- 在 `commit` 阶段需要处理 `effect` 列表，这里的 `effect` 列表包含了根据 `diff` 更新 `DOM` 树、 回调生命周期、 响应 `ref` 等。
- 但一定要注意，这个阶段是同步执行的，不可中断暂停，所以不要在 `componentDidMount`、 `componentDidUpdate`、 `componentWiilUnmount` 中去执行重度消耗算力的任务
- 如果只是一般的应用场景，比如管理后台、H5 展示页等，两者性能差距并不大，但在动画、画布及手势等场景下， `Stack Reconciler` 的设计会占用占主线程，造成卡顿，而 `fiber reconciler` 的设计则能带来高性能的表现

水平垂直居中的实现

- 利用绝对定位，先将元素的左上角通过top:50%和left:50%定位到页面的中心，然后再通过translate来调整元素的中心点到页面的中心。该方法需要**考虑浏览器兼容问题。**

```
.parent {    position: relative;} .child {    position: absolute;    left: 50%;    top: 50%;    trar
```

- 利用绝对定位，设置四个方向的值都为0，并将margin设置为auto，由于宽高固定，因此对应方向实现平分，可以实现水平和垂直方向上的居中。该方法适用于**盒子有宽高**的情况：

```css
.parent {
    position: relative;
}

.child {
    position: absolute;
    top: 0;
    bottom: 0;
    left: 0;
    right: 0;
    margin: auto;
```

```
}
```

- 利用绝对定位，先将元素的左上角通过top:50%和left:50%定位到页面的中心，然后再通过margin负值来调整元素的中心点到页面的中心。该方法适用于**盒子宽高已知**的情况

```css 复制代码
.parent {
    position: relative;
}

.child {
    position: absolute;
    top: 50%;
    left: 50%;
    margin-top: -50px;     /* 自身 height 的一半 */
    margin-left: -50px;    /* 自身 width 的一半 */
}
```

- 使用flex布局，通过align-items:center和justify-content:center设置容器的垂直和水平方向上为居中对齐，然后它的子元素也可以实现垂直和水平的居中。该方法要**考虑兼容的问题**，该方法在移动端用的较多：

```css 复制代码
.parent {
    display: flex;
    justify-content:center;
    align-items:center;
}
```

代码输出结果

```javascript 复制代码
function Person(name) {
    this.name = name
}
var p2 = new Person('king');
console.log(p2.__proto__) //Person.prototype
console.log(p2.__proto__.__proto__) //Object.prototype
console.log(p2.__proto__.__proto__.__proto__) // null
console.log(p2.__proto__.__proto__.__proto__.__proto__)//null后面没有了，报错
console.log(p2.__proto__.__proto__.__proto__.__proto__.__proto__)//null后面没有了，报错
console.log(p2.constructor)//Person
console.log(p2.prototype)//undefined p2是实例，没有prototype属性
```

```
console.log(Person.constructor)//Function 一个空函数
console.log(Person.prototype)//打印出Person.prototype这个对象里所有的方法和属性
console.log(Person.prototype.constructor)//Person
console.log(Person.prototype.__proto__)// Object.prototype
console.log(Person.__proto__) //Function.prototype
console.log(Function.prototype.__proto__)//Object.prototype
console.log(Function.__proto__)//Function.prototype
console.log(Object.__proto__)//Function.prototype
console.log(Object.prototype.__proto__)//null
```

这道义题目考察原型、原型链的基础，记住就可以了。