1. Js连续赋值问题

> 一、连续赋值有顺序
> 比如：A=B=C 它的赋值顺序是 B=C， A=B
> 二、js内部为了保证赋值语句的正确，会在一条赋值语句执行前，先把所有要赋值的引用地址取出一个副本，再依次赋值。

```
//  code 1
let a = {
  n: 1,
};
let b = a;
// b = { n: 1 };
a.x = a = {
  n: 2,
};
// step1 B = C, new* a = { n: 2} //转向新的地址,会预先复制一个副本
// step2 A = B, old* a = { n:1, x:{n:2}}
console.log(a.x); // undefined
console.log(b); // {n: 1, x: {n: 2}}

// code 2
let a = 12,
  b = 12;
function fn() {
  // console.log(a, b); // Uncaught ReferenceError: Cannot access 'a' before initializat
  console.log(b); // 12
  let a = (b = 13);
  console.log(a, b); // 13 13
}
fn();
console.log(a, b); // 12 13

// code 3
let i = 1;
let fn = (i) => (n) => console.log(n + ++i);
let f = fn(1);
f(2); // 4
fn(3)(4); // 8
f(5); // 8
console.log(i); // 1

// code 4
var n = 0;
function a() {
  var n = 10;
  function b() {
    n++;
    console.log(n);
  }
  b();
  return b;
}
var c = a();
c();
console.log(n); // 11 12 0
```

2. 变量提升/静态方法/实例方法/原型方法调用

```
function Foo() {
  getName = function () {
    console.log(1);
  };
  return this;
}
Foo.getName = function () {
  console.log(2);
};
Foo.prototype.getName = function () {
  console.log(3);
};
var getName = function () {
  console.log(4);
};
function getName() {
  console.log(5);
}

Foo.getName(); // 2 解析：调用函数的静态方法
getName(); // 4 解析：函数表达式会覆盖函数声明式方法
Foo().getName(); // 1 解析：调用函数自身的方法
getName(); // 1 解析: 受前一行代码执行影响相当于调用this.getName(), 其中this指向window, 因为Foo
new Foo.getName(); // 2 解析：相当于执行new (Foo.getName)()
new Foo().getName(); // 3 解析：调用函数的实例方法, 相当于执行(new Foo()).getName()
new new Foo().getName(); // 3 解析：相当于执行new (new Foo()).getName()()
// 操作运算符的优先级: () > new > .
```

3. 递归/微任务/宏任务

```
function fn() {
  fn();
}
fn(); // Uncaught RangeError: Maximum call stack size exceeded

var num = 0;
function fn() {
  console.log(num++);
  setTimeout(fn, 1000);
}
fn(); // 可以正常执行，为什么?
// 解析：原因是因为setTImeout属于异步宏任务，不在主线程栈内存中
```

```
console.log("script start"); //1

const promiseA = new Promise((resolve, reject) => {
  console.log("init promiseA"); //2
  resolve("promiseA");
});

const promiseB = new Promise((resolve, reject) => {
  console.log("init promiseB"); //3
  resolve("promiseB");
});

setTimeout(() => {
  console.log("setTimeout run"); //7
  promiseB.then((res) => {
    console.log("promiseB res :>> ", res); //9
  });
  console.log("setTimeout end"); //8
}, 500);

promiseA.then((res) => {
  console.log("promiseA res :>> ", res); //5
});

queueMicrotask(() => {
  console.log("queue Microtask run"); //6
});

console.log("script end"); // 4

// script start
// init promiseA
// init promiseB
// script end
// promiseA res :>>  promiseA
// queue Microtask run
// setTimeout run
// setTimeout end
// promiseB res :>>  promiseB
```

```
const list = [1, 2, 3];
const square = (num) => {
  return new Promise((resolve, reject) => {
    setTimeout(() => {
      resolve(num * num);
    }, 1000);
  });
};

function test() {
  list.forEach(async (x) => {
    const res = await square(x);
    console.log(res);
  });
}
test();

// 执行结果： 1s之后输出 1 4 9

// 不能修改square方法，实现每隔一秒输出结果
const list = [1, 2, 3];
const square = (num) => {
  return new Promise((resolve, reject) => {
    setTimeout(() => {
      resolve(num * num);
    }, 1000);
  });
};

function test() {
  list.forEach((x, index) => {
    setTimeout(async () => {
      const res = await square(x);
      console.log(res);
    }, index * 1000);
  });
}
test();
```

```
console.log(1);
setTimeout(function () {
  console.log(2);
}, 0);
var promise = new Promise(function (resolve, reject) {
  console.log(3);
  setTimeout(function () {
    console.log(4);
    resolve();
  }, 1000);
});
promise.then(function () {
  console.log(5);
  setTimeout(function () {
    console.log(6);
  }, 0);
});
console.log(7);
// 输出结果顺序：1 3 7 2 4 5 6
// 解析：JS代码执行优先级：主线程 -> 微任务 -> 宏任务


var promise = new Promise(function (resolve, reject) {
  setTimeout(function () {
    console.log(1);
    resolve();
  }, 3000);
});
promise
  .then(function () {
    setTimeout(function () {
      console.log(2);
    }, 2000);
  })
  .then(function () {
    setTimeout(function () {
      console.log(3);
    }, 1000);
  })
  .then(function () {
    setTimeout(function () {
      console.log(4);
    }, 0);
  });
// 输出结果：3s后输出1和4，再过1s输出3，再过1s输出2
// 解析：promise.then()方法要等resolve()执行以后，才会执行后面的then方法，后面的这些方法按定时器异步
```

```
async function async1() {
  console.log("async1 start");
  await async2();
  console.log(`async1 end`);
}

async function async2() {
  console.log("async2");
}

console.log("script start");

setTimeout(() => {
  console.log("setTimeout");
}, 0);

async1();

new Promise((resolve, reject) => {
  console.log("promise1");
  resolve();
}).then(() => {
  console.log("promise2");
});

console.log("script end");

// 输出结果:
// script start
// async1 start
// async2
// promise1
// script end
// async1 end
// promise2
// setTimeout
```

promise串行问题（腾讯文档）

```
let promiseArr = [
  () => {
    return new Promise(res => {
      console.log('run 1', Date.now());
      res('run 1 resolve');
    });
  },
  () => {
    return new Promise(res => {
      console.log('run 2', Date.now());
      res('run 2 resolve');
    });
  },
  () => {
    return new Promise(res => {
      console.log('run 3', Date.now());
      res('run 3 resolve');
    });
  },
]

async function fn () {
  for (let i = 0; i < promiseArr.length; i++) {
    // 串行打印console.log;
    // await promiseArr[i]();
    // 串行打印console.log并执行resolve
    await promiseArr[i]().then((value) => {
      console.log(value);
    });
  }
}
fn();
```

事件循环(字节面试题)

```
function test () {
  console.log(1);
  Promise.resolve().then(test);
}
test();
setTimeout(() => {console.log(2)}, 0)

// 打印结果:
// 一直输出1 不会执行setTimeout里面的回调函数
```

promise、async/await

```
// 字节面试题
new Promise((reslove, reject) => {
    reject();
}).then(null, () => {
    console.log(1);
}).then(() => {
    console.log(2);
}).then(() => {
    console.log(3);
});

// 打印结果：  1 2 3


new Promise((reslove, reject) => {
    reject();
}).then(null, () => {
    console.log(1);
}).then(() => {
    new Promise((reslove, reject) => {
        reject();
    }).then(null, () => {
        console.log('a');
    }).then(() => {
        console.log('b');
    }).then(() => {
        console.log('c');
    })
}).then(() => {
    console.log(3);
})

// 打印结果：
// 1
// a
// 3
// b
// c
```

```
new Promise((resolve, reject) => {
    resolve();
}).then(null, () => {
    console.log(1);
}).then(() => {
    new Promise((resolve, reject) => {
        console.log(2);
        resolve();
    }).then(null, () => {
        console.log('a');
    }).then(() => {
        console.log('b');
    }).then(() => {
        console.log('c');
    })
}).then(() => {
    console.log(3);
})

// 打印结果:
// 2
// 3
// b
// c


new Promise((resolve, reject) => {
    reject();
}).then(null, () => {
    console.log(1);
}).then(() => {
    new Promise((resolve, reject) => {
        console.log(2);
        reject();
    }).then(null, () => {
        console.log('a');
    }).then(() => {
        console.log('b');
    }).then(() => {
        console.log('c');
    })
}).then(() => {
    console.log(3);
})
// 1
// 2
// a
// 3
// b
// c
```

```
function foo () {
  var a = 0;
  return function () {
    console.log(a++);
  }
}
var f1 = foo(),
f2 = foo();
f1(); // 0
f1(); // 1
f2(); // 0


function Page() {
  console.log(this);
  return this.hosts;
}
Page.hosts = ['h1'];
Page.prototype.hosts = ['h2'];
var p1 = new Page();
var p2 = Page();
console.log(p1.hosts); // undefined
console.log(p2.hosts); // Uncaught TypeError: Cannot read property 'hosts' of undefined
```

如果让一个不可迭代对象，变成可迭代

```
var obj = {
  0: 0,
  1: 1,
  length: 2,
};
for (i of obj) {
  console.log(i);
}
// 报错：Uncaught TypeError: obj is not iterable

var obj = {
  0: 0,
  1: 1,
  length: 2,
  [Symbol.iterator]: Array.prototype[Symbol.iterator],
};
for (i of obj) {
  console.log(i);
}

// 原理：可迭代对象都拥有@@iterator属性
```

一系列代码题

1.0 异步相关

Q1

```
const promise = new Promise((resolve, reject) => {
  console.log(1);
  console.log(2);
});
promise.then(() => {
  console.log(3);
});
console.log(4);

// 1
// 2
// 4
```

> promise.then 是微任务，它会在所有的宏任务执行完之后才会执行，同时需要 promise 内部的状态发生变化，因为这里内部没有发生变化，一直处于 pending 状态，所以不输出3。

Q2

```
const promise1 = new Promise((resolve, reject) => {
  console.log('promise1')
  resolve('resolve1')
})
const promise2 = promise1.then(res => {
  console.log(res)
})
console.log('1', promise1);
console.log('2', promise2);

// promise1
// 1 Promise{<resolved>: resolve1}
// 2 Promise{<pending>}
// resolve1
```

> 需要注意的是，直接打印promise1，会打印出它的状态值和参数。
>
> - script 是一个宏任务，按照顺序执行这些代码；
> - 首先进入 Promise ，执行该构造函数中的代码，打印 promise1 ；
> - 碰到 resolve 函数, 将 promise1 的状态改变为 resolved , 并将结果保存下来；
> - 碰到 promise1.then 这个微任务，将它放入微任务队列；
> - promise2 是一个新的状态为 pending 的 Promise ；
> - 执行同步代码1， 同时打印出 promise1 的状态是 resolved ；

- 执行同步代码2，同时打印出 `promise2` 的状态是 `pending` ；
- 宏任务执行完毕，查找微任务队列，发现 `promise1.then` 这个微任务且状态为 `resolved` ，执行它。

Q3

```
const promise = new Promise((resolve, reject) => {
  console.log(1);
  setTimeout(() => {
    console.log("timerStart");
    resolve("success");
    console.log("timerEnd");
  }, 0);
  console.log(2);
});
promise.then((res) => {
  console.log(res);
});
console.log(4);

// 1
// 2
// 4
// timerStart
// timerEnd
// success
```

- 首先遇到Promise构造函数，会先执行里面的内容，打印1；
- 遇到定时器steTimeout，它是一个宏任务，放入宏任务队列；
  继续向下执行，打印出2；
- 由于Promise的状态此时还是pending，所以promise.then先不执行；
- 继续执行下面的同步任务，打印出4；
- 此时微任务队列没有任务，继续执行下一轮宏任务，执行steTimeout；
- 首先执行timerStart，然后遇到了resolve，将promise的状态改为resolved且保存结果并将之前的promise.then推入微任务队列，再执行timerEnd；
- 执行完这个宏任务，就去执行微任务promise.then，打印出resolve的结果。

Q4

```
Promise.resolve().then(() => {
  console.log('promise1');
  const timer2 = setTimeout(() => {
    console.log('timer2')
  }, 0)
});
const timer1 = setTimeout(() => {
  console.log('timer1')
  Promise.resolve().then(() => {
    console.log('promise2')
  })
}, 0)
console.log('start');

// start
// promise1
// timer1
// promise2
// timer2
```

- 首先， `Promise.resolve().then` 是一个微任务，加入微任务队列
- 执行 `timer1` ， 它是一个宏任务，加入宏任务队列
- 继续执行下面的同步代码，打印出 `start`
- 这样第一轮宏任务就执行完了，开始执行微任务 `Promise.resolve().then` ， 打印出 `promise1`
- 遇到 `timer2` ， 它是一个宏任务，将其加入宏任务队列，此时宏任务队列有两个任务，分别是 `timer1` 、 `timer2` ；
- 这样第一轮微任务就执行完了，开始执行第二轮宏任务，首先执行定时器timer1，打印 `timer1` ；
- 遇到 `Promise.resolve().then` ， 它是一个微任务，加入微任务队列
- 开始执行微任务队列中的任务，打印 `promise2` ；
- 最后执行宏任务 `timer2` 定时器，打印出 `timer2` ；

Q5

```
const promise1 = new Promise((resolve, reject) => {
  setTimeout(() => {
    resolve('success')
  }, 1000)
})
const promise2 = promise1.then(() => {
  throw new Error('error!!!')
})
console.log('promise1', promise1)
console.log('promise2', promise2)
setTimeout(() => {
  console.log('promise1', promise1)
  console.log('promise2', promise2)
}, 2000)

// promise1 Promise {<pending>}
// promise2 Promise {<pending>}

// Uncaught (in promise) Error: error!!!
// promise1 Promise {<fulfilled>: "success"}
// promise2 Promise {<rejected>: Error: error!!}
```

Q6

```
Promise.resolve().then(() => {
  return new Error('error!!!')
}).then(res => {
  console.log("then: ", res)
}).catch(err => {
  console.log("catch: ", err)
})
// then:  Error: error!!!
```

> 返回任意一个非 promise 的值都会被包裹成 promise 对象，因此这里的 return new Error('error!!!') 也被包裹成了 return Promise.resolve(new Error('error!!!'))，因此它会被 then 捕获而不是 catch 。

Q7 finally

```
Promise.resolve('1')
  .then(res => {
    console.log(res)
  })
  .finally(() => {
    console.log('finally')
  })
Promise.resolve('2')
  .finally(() => {
    console.log('finally2')
        return '我是finally2返回的值'
  })
  .then(res => {
    console.log('finally2后面的then函数', res)
  })

// 1
// finally2
// finally
// finally2后面的then函数 2
```

> - `.finally()` 一般用的很少，只要记住以下几点就可以了：
> - `.finally()` 方法不管Promise对象最后的状态如何都会执行
> - `.finally()` 方法的回调函数不接受任何的参数，也就是说你在 `.finally()` 函数中是无法知道 `Promise` 最终的状态是 `resolved` 还是 `rejected` 的
> - 它最终返回的默认会是一个上一次的 `Promise` 对象值，不过如果抛出的是一个异常则返回异常的 `Promise` 对象。
> - `finally` 本质上是 `then` 方法的特例

Q8

```
function runAsync(x) {
  const p = new Promise(r => setTimeout(() => r(x, console.log(x)), 1000))
  return p
}
function runReject(x) {
  const p = new Promise((res, rej) => setTimeout(() => rej(`Error: ${x}`, console.log(x
  return p
}
Promise.all([runAsync(1), runReject(4), runAsync(3), runReject(2)])
  .then(res => console.log(res))
  .catch(err => console.log(err))

// // 1s后输出
// 1
// 3
// // 2s后输出
// 2
// Error: 2
// // 4s后输出
// 4

Promise.race([runAsync(1), runAsync(2), runAsync(3)])
  .then(res => console.log('result: ', res))
  .catch(err => console.log(err))
// 1
// 'result: ' 1
// 2
// 3
```

> - `.catch` 捕获到了第一个错误，在这道题目中最先的错误就是 `runReject(2)` 的结果。
> - 如果一组异步操作中有一个异常都不会进入 `.then()` 的第一个回调函数参数中。会被 `.then()` 的第二个回调函数捕获

> - `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。

Q9

```
async function async1() {
  console.log("async1 start");
  await async2();
  console.log("async1 end");
}
async function async2() {
  console.log("async2");
}
async1();
console.log('start')
// async1 start
// async2
// start
// async1 end
```

> 代码的执行过程如下：
>
> - 首先执行函数中的同步代码 `async1 start` ， 之后遇到了 `await` ， 它会阻塞 `async1` 后面代码的执行，因此会先去执行 `async2` 中的同步代码 `async2` ， 然后跳出 `async1` ；
> - **跳出 `async1` 函数后，执行同步代码 `start`** ；
> - 在一轮宏任务全部执行完之后，再来执行await后面的内容async1 end。
>   这里可以理解为await后面的语句相当于放到了new Promise中，下一行及之后的语句相当于放在Promise.then中。

Q10

```
async function async1() {
  console.log("async1 start");
  await async2();
  console.log("async1 end");
  setTimeout(() => {
    console.log('timer1')
  }, 0)
}
async function async2() {
  setTimeout(() => {
    console.log('timer2')
  }, 0)
  console.log("async2");
}
async1();
setTimeout(() => {
  console.log('timer3')
}, 0)
console.log("start")
// async1 start
// async2
// start
// async1 end
// timer2
// timer3
// timer1
```

> - 首先进入 async1 ， 打印出 async1 start ；
> - 之后遇到 async2 ， 进入 async2 ， 遇到定时器 timer2 ， 加入宏任务队列，之后打印 async2 ；
> - 由于 async2 阻塞了后面代码的执行，所以执行后面的定时器 timer3 ， 将其加入宏任务队列，之后打印 start ；
> - 然后执行 async2 后面的代码，打印出 async1 end ， 遇到定时器 timer1 ， 将其加入宏任务队列；
> - 最后，宏任务队列有三个任务，先后顺序为 timer2 ， timer3 ， timer1 ， 没有微任务，所以直接所有的宏任务按照先进先出的原则执行。

// 看到20题

2.0 this指向

Q1

```
var a = 10
var obj = {
  a: 20,
  say: () => {
    console.log(this.a)
  }
}
obj.say()

var anotherObj = { a: 30 }
obj.say.apply(anotherObj)
// 10
// 10
```

> 箭头函数时不绑定this的，它的this来自原其父级所处的上下文，所以首先会打印全局中的 a 的值10。后面虽然让say方法指向了另外一个对象，但是仍不能改变箭头函数的特性，它的this仍然是指向全局的，所以依旧会输出10

Q2

```
var obj = {
  name : 'cuggz',
  fun : function(){
    console.log(this.name);
  }
}
obj.fun()     // cuggz
new obj.fun() // undefined
```

> 使用 `new` 构造函数时，其 `this` 指向的是全局环境 `window`

Q3

```
var obj = {
   say: function() {
     var f1 = () =>  {
       console.log("1111", this);
     }
     f1();
   },
   pro: {
     getPro:() =>  {
        console.log(this);
     }
   }
}
var o = obj.say;
o();
obj.say();
obj.pro.getPro();
// 1111 window对象
// 1111 obj对象
// window对象
```

1. `o()`，o是在全局执行的，而f1是箭头函数，它是没有绑定this的，它的this指向其父级的this，其父级say方法的this指向的是全局作用域，所以会打印出window；
2. `obj.say()`，谁调用say，say 的this就指向谁，所以此时this指向的是obj对象；
3. `obj.pro.getPro()`，我们知道，箭头函数时不绑定this的，getPro处于pro中，而对象不构成单独的作用域，所以箭头的函数的this就指向了全局作用域window。

Q4

```
var length = 10;
function fn() {
    console.log(this.length);
}

var obj = {
  length: 5,
  method: function(fn) {
    fn();
    arguments[0]();
  }
};

obj.method(fn, 1);
// 10 2
```

1. 第一次执行fn()，this指向window对象，输出10。
2. 第二次执行arguments0，相当于arguments调用方法，this指向arguments，而这里传了两个参数，故输出arguments长度为2。

Q5

```
var a = 1;
function printA(){
  console.log(this.a);
}
var obj={
  a:2,
  foo:printA,
  bar:function(){
    printA();
  }
}

obj.foo(); // 2
obj.bar(); // 1
var foo = obj.foo;
foo(); // 1
```

- `obj.foo()`，foo 的this指向obj对象，所以a会输出2；
- `obj.bar()`，printA在bar方法中执行，所以此时printA的this指向的是window，所以会输出1；
- `foo()`，foo是在全局对象中执行的，所以其this指向的是window，所以会输出1；

Q6

```
var x = 3;
var y = 4;
var obj = {
    x: 1,
    y: 6,
    getX: function() {
        var x = 5;
        return function() {
            return this.x;
        }();
    },
    getY: function() {
        var y = 7;
        return this.y;
    }
}
console.log(obj.getX()) // 3
console.log(obj.getY()) // 6
```

> - **匿名函数**的this是指向全局对象的，所以this指向window，会打印出3；
> - getY是由obj调用的，所以其this指向的是obj对象，会打印出6。

> this绑定的优先级：**new绑定 > 显式绑定 > 隐式绑定 > 默认绑定**。

3.0 作用域&变量提升&闭包

4.0 原型&继承