

# 「2021」高频前端面试题汇总之JavaScript篇（下）

## 六、this/call/apply/bind

---

### 1. 对this对象的理解

this 是执行上下文中的一个属性，它指向最后一次调用这个方法的对象。在实际开发中，this 的指向可以通过四种调用模式来判断。

- 第一种是**函数调用模式**，当一个函数不是一个对象的属性时，直接作为函数来调用时，this 指向全局对象。
- 第二种是**方法调用模式**，如果一个函数作为一个对象的方法来调用时，this 指向这个对象。
- 第三种是**构造器调用模式**，如果一个函数用 new 调用时，函数执行前会新创建一个对象，this 指向这个新创建的对象。
- 第四种是 **apply、call 和 bind 调用模式**，这三个方法都可以显示的指定调用函数的 this 指向。其中 apply 方法接收两个参数：一个是 this 绑定的对象，一个是参数数组。call 方法接收的参数，第一个是 this 绑定的对象，后面的其余参数是传入函数执行的参数。也就是说，在使用 call() 方法时，传递给函数的参数必须逐个列举出来。bind 方法通过传入一个对象，返回一个 this 绑定了传入对象的新函数。这个函数的 this 指向除了使用 new 时会被改变，其他情况下都不会改变。

这四种方式，使用构造器调用模式的优先级最高，然后是 apply、call 和 bind 调用模式，然后是方法调用模式，然后是函数调用模式。

### 2. call() 和 apply() 的区别？

它们的作用一模一样，区别仅在于传入参数的形式的不同。

- apply 接受两个参数，第一个参数指定了函数体内 this 对象的指向，第二个参数为一个带下标的集合，这个集合可以为数组，也可以为类数组，apply 方法把这个集合中的元素作为参数传递给被调用的函数。
- call 传入的参数数量不固定，跟 apply 相同的是，第一个参数也是代表函数体内的 this 指向，从第二个参数开始往后，每个参数被依次传入函数。

### 3. 实现call、apply 及 bind 函数

#### (1) call 函数的实现步骤:

- 判断调用对象是否为函数，即使是定义在函数的原型上的，但是可能出现使用 call 等方式调用的情况。
- 判断传入上下文对象是否存在，如果不存在，则设置为 window 。
- 处理传入的参数，截取第一个参数后的所有参数。
- 将函数作为上下文对象的一个属性。
- 使用上下文对象来调用这个方法，并保存返回结果。
- 删除刚才新增的属性。
- 返回结果。

javascript 复制代码

```
Function.prototype.myCall = function(context) {  
  // 判断调用对象  
  if (typeof this !== "function") {  
    console.error("type error");  
  }  
  // 获取参数  
  let args = [...arguments].slice(1),  
      result = null;  
  // 判断 context 是否传入，如果未传入则设置为 window  
  context = context || window;  
  // 将调用函数设为对象的方法  
  context.fn = this;  
  // 调用函数  
  result = context.fn(...args);  
  // 将属性删除  
  delete context.fn;  
  return result;  
};
```

#### (2) apply 函数的实现步骤:

- 判断调用对象是否为函数，即使是定义在函数的原型上的，但是可能出现使用 call 等方式调用的情况。
- 判断传入上下文对象是否存在，如果不存在，则设置为 window 。
- 将函数作为上下文对象的一个属性。
- 判断参数值是否传入
- 使用上下文对象来调用这个方法，并保存返回结果。
- 删除刚才新增的属性
- 返回结果

```

Function.prototype.myApply = function(context) {
  // 判断调用对象是否为函数
  if (typeof this !== "function") {
    throw new TypeError("Error");
  }
  let result = null;
  // 判断 context 是否存在，如果未传入则为 window
  context = context || window;
  // 将函数设为对象的方法
  context.fn = this;
  // 调用方法
  if (arguments[1]) {
    result = context.fn(...arguments[1]);
  } else {
    result = context.fn();
  }
  // 将属性删除
  delete context.fn;
  return result;
};

```

### (3) bind 函数的实现步骤:

- 判断调用对象是否为函数，即使是定义在函数的原型上的，但是可能出现使用 call 等方式调用的情况。
- 保存当前函数的引用，获取其余传入参数值。
- 创建一个函数返回
- 函数内部使用 apply 来绑定函数调用，需要判断函数作为构造函数的情况，这个时候需要传入当前函数的 this 给 apply 调用，其余情况都传入指定的上下文对象。

```

Function.prototype.myBind = function(context) {
  // 判断调用对象是否为函数
  if (typeof this !== "function") {
    throw new TypeError("Error");
  }
  // 获取参数
  var args = [...arguments].slice(1),
      fn = this;
  return function Fn() {
    // 根据调用方式，传入不同绑定值
    return fn.apply(
      this instanceof Fn ? this : context,
      args.concat(...arguments)
    );
  };
}

```

```
};  
};
```

## 七、异步编程

---

### 1. 异步编程的实现方式？

JavaScript中的异步机制可以分为以下几种：

- **回调函数** 的方式，使用回调函数的方式有一个缺点是，多个回调函数嵌套的时候会造成回调函数地狱，上下两层的回调函数间的代码耦合度太高，不利于代码的可维护。
- **Promise** 的方式，使用 Promise 的方式可以将嵌套的回调函数作为链式调用。但是使用这种方法，有时会造成多个 then 的链式调用，可能会造成代码的语义不够明确。
- **generator** 的方式，它可以在函数的执行过程中，将函数的执行权转移出去，在函数外部还可以将执行权转移回来。当遇到异步函数执行的时候，将函数执行权转移出去，当异步函数执行完毕时再将执行权给转移回来。因此在 generator 内部对于异步操作的方式，可以以同步的顺序来书写。使用这种方式需要考虑的问题是何时将函数的控制权转移回来，因此需要有一个自动执行 generator 的机制，比如说 co 模块等方式来实现 generator 的自动执行。
- **async 函数** 的方式，async 函数是 generator 和 promise 实现的一个自动执行的语法糖，它内部自带执行器，当函数内部执行到一个 await 语句的时候，如果语句返回一个 promise 对象，那么函数将会等待 promise 对象的状态变为 resolve 后再继续向下执行。因此可以将异步逻辑，转化为同步的顺序来书写，并且这个函数可以自动执行。

### 2. setTimeout、Promise、Async/Await 的区别

#### (1) setTimeout

```
console.log('script start')    //1. 打印 script start  
setTimeout(function(){  
    console.log('settimeout')  // 4. 打印 settimeout  
})    // 2. 调用 setTimeout 函数，并定义其完成后执行的回调函数  
console.log('script end')      //3. 打印 script start  
// 输出顺序: script start->script end->settimeout
```

javascript 复制代码

#### (2) Promise

Promise本身是**同步的立即执行函数**， 当在executor中执行resolve或者reject的时候, 此时是异步操作， 会先执行then/catch等， 当主栈完成后， 才会去调用resolve/reject中存放的方法执行， 打印p的时候， 是打印的返回结果， 一个Promise实例。

javascript 复制代码

```
console.log('script start')
let promise1 = new Promise(function (resolve) {
  console.log('promise1')
  resolve()
  console.log('promise1 end')
}).then(function () {
  console.log('promise2')
})
setTimeout(function(){
  console.log('settimeout')
})
console.log('script end')
// 输出顺序: script start->promise1->promise1 end->script end->promise2->settimeout
```

当JS主线程执行到Promise对象时：

- promise1.then() 的回调就是一个 task
- promise1 是 resolved或rejected: 那这个 task 就会放入当前事件循环回合的 microtask queue
- promise1 是 pending: 这个 task 就会放入 事件循环的未来的某个(可能下一个)回合的 microtask queue 中
- setTimeout 的回调也是个 task， 它会被放入 macrotask queue 即使是 0ms 的情况

### (3) async/await

javascript 复制代码

```
async function async1(){
  console.log('async1 start');
  await async2();
  console.log('async1 end')
}
async function async2(){
  console.log('async2')
}
console.log('script start');
async1();
console.log('script end')
// 输出顺序: script start->async1 start->async2->script end->async1 end
```

async 函数返回一个 Promise 对象，当函数执行的时候，一旦遇到 await 就会先返回，等到触发的异步操作完成，再执行函数体内后面的语句。可以理解为，是让出了线程，跳出了 async 函数体。

例如：

javascript 复制代码

```
async function func1() {  
  return 1  
}  
console.log(func1())
```

```
▼ Promise {<resolved>: 1} ⓘ  
  ► __proto__: Promise  
    [[PromiseStatus]]: "resolved"  
    [[PromiseValue]]: 1
```

@稀土掘金技术社区

func1的运行结果其实就是一个Promise对象。因此也可以使用then来处理后续逻辑。

javascript 复制代码

```
func1().then(res => {  
  console.log(res); // 30  
})
```

await的含义为等待，也就是 async 函数需要等待await后的函数执行完成并且有了返回结果（Promise对象）之后，才能继续执行下面的代码。await通过返回一个Promise对象来实现同步的效果。

### 3. 对Promise的理解

Promise是异步编程的一种解决方案，它是一个对象，可以获取异步操作的消息，他的出现大大改善了异步编程的困境，避免了地狱回调，它比传统的解决方案回调函数和事件更合理和更强大。

所谓Promise，简单说就是一个容器，里面保存着某个未来才会结束的事件（通常是一个异步操作）的结果。从语法上说，Promise 是一个对象，从它可以获取异步操作的消息。Promise 提供统一的 API，各种异步操作都可以用同样的方法进行处理。

(1) Promise的实例有三个状态:

- Pending（进行中）

- Resolved (已完成)
- Rejected (已拒绝)

当把一件事情交给promise时，它的状态就是Pending，任务完成了状态就变成了Resolved、没有完成失败了就变成了Rejected。

(2) Promise的实例有**两个过程**：

- pending -> fulfilled : Resolved (已完成)
- pending -> rejected: Rejected (已拒绝)

注意：一旦从进行状态变成为其他状态就永远不能更改状态了。

### Promise的特点：

- 对象的状态不受外界影响。promise对象代表一个异步操作，有三种状态，`pending`（进行中）、`fulfilled`（已成功）、`rejected`（已失败）。只有异步操作的结果，可以决定当前是哪一种状态，任何其他操作都无法改变这个状态，这也是promise这个名字的由来——“**承诺**”；
- 一旦状态改变就不会再变，任何时候都可以得到这个结果。promise对象的状态改变，只有两种可能：从 `pending` 变为 `fulfilled`，从 `pending` 变为 `rejected`。这时就称为 `resolved`（已定型）。如果改变已经发生了，你再对promise对象添加回调函数，也会立即得到这个结果。这与事件（event）完全不同，事件的特点是：如果你错过了它，再去监听是得不到结果的。

### Promise的缺点：

- 无法取消Promise，一旦新建它就会立即执行，无法中途取消。
- 如果不设置回调函数，Promise内部抛出的错误，不会反应到外部。
- 当处于pending状态时，无法得知目前进展到哪一个阶段（刚刚开始还是即将完成）。

**总结：** Promise 对象是异步编程的一种解决方案，最早由社区提出。Promise 是一个构造函数，接收一个函数作为参数，返回一个 Promise 实例。一个 Promise 实例有三种状态，分别是pending、resolved 和 rejected，分别代表了进行中、已成功和已失败。实例的状态只能由pending 转变 resolved 或者rejected 状态，并且状态一经改变，就凝固了，无法再被改变了。

状态的改变是通过 `resolve()` 和 `reject()` 函数来实现的，可以在异步操作结束后调用这两个函数改变 Promise 实例的状态，它的原型上定义了一个 `then` 方法，使用这个 `then` 方法可以为两

个状态的改变注册回调函数。这个回调函数属于微任务，会在本轮事件循环的末尾执行。

**注意：** 在构造 `Promise` 的时候，构造函数内部的代码是立即执行的

## 4. Promise的基本用法

### (1) 创建Promise对象

Promise对象代表一个异步操作，有三种状态：pending（进行中）、fulfilled（已成功）和 rejected（已失败）。

Promise构造函数接受一个函数作为参数，该函数的两个参数分别是 `resolve` 和 `reject`。

javascript 复制代码

```
const promise = new Promise(function(resolve, reject) {  
  // ... some code  
  if (/* 异步操作成功 */) {  
    resolve(value);  
  } else {  
    reject(error);  
  }  
});
```

一般情况下都会使用 `new Promise()` 来创建promise对象，但是也可以使用 `promise.resolve` 和 `promise.reject` 这两个方法：

- **Promise.resolve**

`Promise.resolve(value)` 的返回值也是一个promise对象，可以对返回值进行.then调用，代码如下：

javascript 复制代码

```
Promise.resolve(11).then(function(value){  
  console.log(value); // 打印出11  
});
```

`resolve(11)` 代码中，会让promise对象进入确定( `resolve` 状态)，并将参数 `11` 传递给后面的 `then` 所指定的 `onFulfilled` 函数；

创建promise对象可以使用 `new Promise` 的形式创建对象，也可以使用 `Promise.resolve(value)` 的形式创建promise对象；



- **Promise.reject**

`Promise.reject` 也是 `new Promise` 的快捷形式，也创建一个promise对象。代码如下：

javascript 复制代码

```
Promise.reject(new Error("我错了，请原谅俺！！"));
```

就是下面的代码new Promise的简单形式：

javascript 复制代码

```
new Promise(function(resolve,reject){  
    reject(new Error("我错了！"));  
});
```

下面是使用resolve方法和reject方法：

javascript 复制代码

```
function testPromise(ready) {  
    return new Promise(function(resolve,reject){  
        if(ready) {  
            resolve("hello world");  
        }else {  
            reject("No thanks");  
        }  
    });  
};  
// 方法调用  
testPromise(true).then(function(msg){  
    console.log(msg);  
},function(error){  
    console.log(error);  
});
```

上面的代码的含义是给 `testPromise` 方法传递一个参数，返回一个promise对象，如果为 `true` 的话，那么调用promise对象中的 `resolve()` 方法，并且把其中的参数传递给后面的 `then` 第一个函数内，因此打印出 “ `hello world` ”，如果为 `false` 的话，会调用promise对象中的 `reject()` 方法，则会进入 `then` 的第二个函数内，会打印 `No thanks` ；

## (2) Promise方法

Promise有五个常用的方法：`then()`、`catch()`、`all()`、`race()`、`finally`。下面就来看一下这些方法。

## 1. then()

当Promise执行的内容符合成功条件时，调用 `resolve` 函数，失败就调用 `reject` 函数。Promise创建完了，那该如何调用呢？

javascript 复制代码

```
promise.then(function(value) {  
  // success  
}, function(error) {  
  // failure  
});
```

`then` 方法可以接受两个回调函数作为参数。第一个回调函数是Promise对象的状态变为 `resolved` 时调用，第二个回调函数是Promise对象的状态变为 `rejected` 时调用。其中第二个参数可以省略。`then` 方法返回的是一个新的Promise实例（不是原来那个Promise实例）。因此可以采用链式写法，即 `then` 方法后面再调用另一个`then`方法。

当要写有顺序的异步事件时，需要串行时，可以这样写：

javascript 复制代码

```
let promise = new Promise((resolve, reject) => {  
  ajax('first').success(function(res) {  
    resolve(res);  
  })  
})  
promise.then(res => {  
  return new Promise((resovle, reject) => {  
    ajax('second').success(function(res) {  
      resolve(res)  
    })  
  })  
}).then(res => {  
  return new Promise((resovle, reject) => {  
    ajax('second').success(function(res) {  
      resolve(res)  
    })  
  })  
}).then(res => {  
  
})
```

那当要写的事件没有顺序或者关系时，还如何写呢？可以使用 `all` 方法来解决。

## 2. catch()

Promise对象除了有then方法，还有一个catch方法，该方法相当于 then 方法的第二个参数，指向 reject 的回调函数。不过 catch 方法还有一个作用，就是在执行 resolve 回调函数时，如果出现错误，抛出异常，不会停止运行，而是进入 catch 方法中。

javascript 复制代码

```
p.then((data) => {
    console.log('resolved',data);
},(err) => {
    console.log('rejected',err);
});
p.then((data) => {
    console.log('resolved',data);
}).catch((err) => {
    console.log('rejected',err);
});
```

### 3. all()

all 方法可以完成并行任务，它接收一个数组，数组的每一项都是一个 promise 对象。当数组中所有的 promise 的状态都达到 resolved 的时候，all 方法的状态就会变成 resolved，如果有一个状态变成了 rejected，那么 all 方法的状态就会变成 rejected。

javascript 复制代码

```
javascript
let promise1 = new Promise((resolve,reject)=>{
    setTimeout(()=>{
        resolve(1);
    },2000)
});
let promise2 = new Promise((resolve,reject)=>{
    setTimeout(()=>{
        resolve(2);
    },1000)
});
let promise3 = new Promise((resolve,reject)=>{
    setTimeout(()=>{
        resolve(3);
    },3000)
});
Promise.all([promise1,promise2,promise3]).then(res=>{
    console.log(res);
    //结果为: [1,2,3]
})
```

调用 `all` 方法时的结果成功的时候是回调函数的参数也是一个数组，这个数组按顺序保存着每一个 `promise` 对象 `resolve` 执行时的值。

#### (4) `race()`

`race` 方法和 `all` 一样，接受的参数是一个每项都是 `promise` 的数组，但是与 `all` 不同的是，当最先执行完的事件执行完之后，就直接返回该 `promise` 对象的值。如果第一个 `promise` 对象状态变成 `resolved`，那自身的状态变成了 `resolved`；反之第一个 `promise` 变成 `rejected`，那自身状态就会变成 `rejected`。

javascript 复制代码

```
let promise1 = new Promise((resolve, reject) => {
    setTimeout(() => {
        reject(1);
    }, 2000)
});
let promise2 = new Promise((resolve, reject) => {
    setTimeout(() => {
        resolve(2);
    }, 1000)
});
let promise3 = new Promise((resolve, reject) => {
    setTimeout(() => {
        resolve(3);
    }, 3000)
});
Promise.race([promise1, promise2, promise3]).then(res => {
    console.log(res);
    // 结果: 2
}, rej => {
    console.log(rej);
})
```

那么 `race` 方法有什么实际作用呢？当要做一件事，超过多长时间就不做了，可以用这个方法来解决：

javascript 复制代码

```
Promise.race([promise1, timeoutPromise(5000)]).then(res => {})
```

#### 5. `finally()`

`finally` 方法用于指定不管 `Promise` 对象最后状态如何，都会执行的操作。该方法是 ES2018 引入标准的。

```
promise
  .then(result => {...})
  .catch(error => {...})
  .finally(() => {...});
```

上面代码中，不管 `promise` 最后的状态，在执行完 `then` 或 `catch` 指定的回调函数以后，都会执行 `finally` 方法指定的回调函数。

下面是一个例子，服务器使用 `Promise` 处理请求，然后使用 `finally` 方法关掉服务器。

```
server.listen(port)
  .then(function () {
    // ...
  })
  .finally(server.stop);
```

`finally` 方法的回调函数不接受任何参数，这意味着没有办法知道，前面的 `Promise` 状态到底是 `fulfilled` 还是 `rejected`。这表明，`finally` 方法里面的操作，应该是与状态无关的，不依赖于 `Promise` 的执行结果。`finally` 本质上是 `then` 方法的特例：

```
promise
  .finally(() => {
    // 语句
  });
// 等同于
promise
  .then(
    result => {
      // 语句
      return result;
    },
    error => {
      // 语句
      throw error;
    }
  );
```

上面代码中，如果不使用 `finally` 方法，同样的语句需要为成功和失败两种情况各写一次。有了 `finally` 方法，则只需要写一次。

## 5. Promise解决了什么问题

在工作中经常会碰到这样一个需求，比如我使用ajax发一个A请求后，成功后拿到数据，需要把数据传给B请求；那么需要如下编写代码：

javascript 复制代码

```
let fs = require('fs')
fs.readFile('./a.txt', 'utf8', function(err, data){
  fs.readFile(data, 'utf8', function(err, data){
    fs.readFile(data, 'utf8', function(err, data){
      console.log(data)
    })
  })
})
```

上面的代码有如下缺点：

- 后一个请求需要依赖于前一个请求成功后，将数据往下传递，会导致多个ajax请求嵌套的情况，代码不够直观。
- 如果前后两个请求不需要传递参数的情况下，那么后一个请求也需要前一个请求成功后再执行下一步操作，这种情况下，那么也需要如上编写代码，导致代码不够直观。

Promise 出现之后，代码变成这样：

javascript 复制代码

```
let fs = require('fs')
function read(url){
  return new Promise((resolve, reject)=>{
    fs.readFile(url, 'utf8', function(error, data){
      error && reject(error)
      resolve(data)
    })
  })
}
read('./a.txt').then(data=>{
  return read(data)
}).then(data=>{
  return read(data)
}).then(data=>{
  console.log(data)
})
```

这样代码看起了就简洁了很多，解决了地狱回调的问题。

## 6. Promise.all和Promise.race的区别的使用场景

**(1) Promise.all** `Promise.all` 可以将多个 `Promise` 实例包装成一个新的`Promise`实例。同时，成功和失败的返回值是不同的，成功的时候返回的是一个**结果数组**，而失败的时候则返回**最先被reject失败状态的值**。

`Promise.all`中传入的是数组，返回的也是数组，并且会将进行映射，传入的`promise`对象返回的值是按照顺序在数组中排列的，但是注意的是他们执行的顺序并不是按照顺序的，除非可迭代对象为空。

需要注意，`Promise.all`获得的成功结果的数组里面的数据顺序和`Promise.all`接收到的数组顺序是一致的，这样当遇到发送多个请求并根据请求顺序获取和使用数据的场景，就可以使用`Promise.all`来解决。

## (2) Promise.race

顾名思义，`Promise.race`就是赛跑的意思，意思就是说，`Promise.race([p1, p2, p3])`里面哪个结果获得的快，就返回那个结果，不管结果本身是成功状态还是失败状态。当要做一件事，超过多长时间就不做了，可以用这个方法来解决：

```
Promise.race([promise1, timeoutPromise(5000)]).then(res=>{})
```

javascript 复制代码

## 7. 对async/await 的理解

`async/await`其实是 `Generator` 的语法糖，它能实现的效果都能用`then`链来实现，它是为优化`then`链而开发出来的。从字面上来看，`async`是“异步”的简写，`await`则为等待，所以很好理解`async` 用于申明一个 `function` 是异步的，而 `await` 用于等待一个异步方法执行完成。当然语法上强制规定`await`只能出现在`async`函数中，先来看看`async`函数返回了什么：

```
async function testAsy(){
  return 'hello world';
}
let result = testAsy();
console.log(result)
```

javascript 复制代码



所以，`async` 函数返回的是一个 `Promise` 对象。`async` 函数（包含函数语句、函数表达式、`Lambda`表达式）会返回一个 `Promise` 对象，如果在函数中 `return` 一个直接量，`async` 会把这个直接量通过 `Promise.resolve()` 封装成 `Promise` 对象。

`async` 函数返回的是一个 `Promise` 对象，所以在最外层不能用 `await` 获取其返回值的条件下，当然应该用原来的方式：`then()` 链来处理这个 `Promise` 对象，就像这样：

javascript 复制代码

```
async function testAsy(){
  return 'hello world'
}
let result = testAsy()
console.log(result)
result.then(v=>{
  console.log(v) // hello world
})
```

那如果 `async` 函数没有返回值，又该如何？很容易想到，它会返回

`Promise.resolve(undefined)`。

联想一下 `Promise` 的特点——无等待，所以在没有 `await` 的情况下执行 `async` 函数，它会立即执行，返回一个 `Promise` 对象，并且，绝不会阻塞后面的语句。这和普通返回 `Promise` 对象的函数并无二致。

**注意：** `Promise.resolve(x)` 可以看作是 `new Promise(resolve => resolve(x))` 的简写，可以用于快速封装字面量对象或其他对象，将其封装成 `Promise` 实例。

## 8. `await` 到底在等啥？

**`await` 在等待什么呢？** 一般来说，都认为 `await` 是在等待一个 `async` 函数完成。不过按语法说明，`await` 等待的是一个表达式，这个表达式的计算结果是 `Promise` 对象或者其它值（换句话说，就是没有特殊限定）。



因为 `async` 函数返回一个 `Promise` 对象，所以 `await` 可以用于等待一个 `async` 函数的返回值——这也可以说是 `await` 在等 `async` 函数，但要清楚，它等的实际是一个返回值。注意到 `await` 不仅仅用于等 `Promise` 对象，它可以等任意表达式的结果，所以，`await` 后面实际是可以接普通函数调用或者直接量的。所以下面这个示例完全可以正确运行：

javascript 复制代码

```
function getSomething() {
    return "something";
}
async function testAsync() {
    return Promise.resolve("hello async");
}
async function test() {
    const v1 = await getSomething();
    const v2 = await testAsync();
    console.log(v1, v2);
}
test();
```

`await` 表达式的运算结果取决于它等的是什么。

- 如果它等到的不是一个 `Promise` 对象，那 `await` 表达式的运算结果就是它等到的东西。
- 如果它等到的是一个 `Promise` 对象，`await` 就忙起来了，它会阻塞后面的代码，等着 `Promise` 对象 `resolve`，然后得到 `resolve` 的值，作为 `await` 表达式的运算结果。

来看一个例子：

javascript 复制代码

```
function testAsy(x){
    return new Promise(resolve=>{setTimeout(() => {
        resolve(x);
    }, 3000)
    })
}
async function testAwt() {
    let result = await testAsy('hello world');
    console.log(result);    // 3秒钟之后出现hello world
    console.log('cuger')   // 3秒钟之后出现cug
}
testAwt();
console.log('cug')    //立即输出cug
```

这就是 `await` 必须用在 `async` 函数中的原因。`async` 函数调用不会造成阻塞，它内部所有的阻塞都被封装在一个 `Promise` 对象中异步执行。`await` 暂停当前 `async` 的执行，所以 'cug' 最先输

出，hello world'和'cuger'是3秒钟后同时出现的。

## 9. async/await的优势

单一的 Promise 链并不能发现 async/await 的优势，但是，如果需要处理由多个 Promise 组成的 then 链的时候，优势就能体现出来了（很有意思，Promise 通过 then 链来解决多层回调的问题，现在又用 async/await 来进一步优化它）。

假设一个业务，分多个步骤完成，每个步骤都是异步的，而且依赖于上一个步骤的结果。仍然用 `setTimeout` 来模拟异步操作：

javascript 复制代码

```
/**
 * 传入参数 n，表示这个函数执行的时间（毫秒）
 * 执行的结果是 n + 200，这个值将用于下一步骤
 */
function takeLongTime(n) {
  return new Promise(resolve => {
    setTimeout(() => resolve(n + 200), n);
  });
}

function step1(n) {
  console.log(`step1 with ${n}`);
  return takeLongTime(n);
}

function step2(n) {
  console.log(`step2 with ${n}`);
  return takeLongTime(n);
}

function step3(n) {
  console.log(`step3 with ${n}`);
  return takeLongTime(n);
}
```

现在用 Promise 方式来实现这三个步骤的处理：

javascript 复制代码

```
function doIt() {
  console.time("doIt");
  const time1 = 300;
  step1(time1)
    .then(time2 => step2(time2))
    .then(time3 => step3(time3))
    .then(result => {
      console.log(`result is ${result}`);
      console.timeEnd("doIt");
    });
}
```

```

    });
}
doIt();
// c:\var\test>node --harmony_async_await .
// step1 with 300
// step2 with 500
// step3 with 700
// result is 900
// doIt: 1507.251ms

```

输出结果 `result` 是 `step3()` 的参数  $700 + 200 = 900$ 。`doIt()` 顺序执行了三个步骤，一共用了  $300 + 500 + 700 = 1500$  毫秒，和 `console.time()/console.timeEnd()` 计算的结果一致。

如果用 `async/await` 来实现呢，会是这样：

javascript 复制代码

```

async function doIt() {
    console.time("doIt");
    const time1 = 300;
    const time2 = await step1(time1);
    const time3 = await step2(time2);
    const result = await step3(time3);
    console.log(`result is ${result}`);
    console.timeEnd("doIt");
}
doIt();

```

结果和之前的 `Promise` 实现是一样的，但是这个代码看起来是不是清晰得多，几乎跟同步代码一样

## 10. async/await对比Promise的优势

- 代码读起来更加同步，`Promise`虽然摆脱了回调地狱，但是`then`的链式调用也会带来额外的阅读负担
- `Promise`传递中间值非常麻烦，而`async/await`几乎是同步的写法，非常优雅
- 错误处理友好，`async/await`可以用成熟的`try/catch`，`Promise`的错误捕获非常冗余
- 调试友好，`Promise`的调试很差，由于没有代码块，你不能在一个返回表达式的箭头函数中设置断点，如果你在一个`.then`代码块中使用调试器的步进(step-over)功能，调试器并不会进入后续的`.then`代码块，因为调试器只能跟踪同步代码的每一步。

## 11. async/await 如何捕获异常

```
async function fn(){
  try{
    let a = await Promise.reject('error')
  }catch(error){
    console.log(error)
  }
}
```

## 八、面向对象

---

### 1. 对象创建的方式有哪些？

一般使用字面量的形式直接创建对象，但是这种创建方式对于创建大量相似对象的时候，会产生大量的重复代码。但 js 和一般的面向对象的语言不同，在 ES6 之前它没有类的概念。但是可以使用函数来进行模拟，从而产生出可复用的对象创建方式，常见的有以下几种：

(1) 第一种是工厂模式，工厂模式的主要工作原理是用函数来封装创建对象的细节，从而通过调用函数来达到复用的目的。但是它有一个很大的问题就是创建出来的对象无法和某个类型联系起来，它只是简单的封装了复用代码，而没有建立起对象和类型间的关系。

(2) 第二种是构造函数模式。js 中每一个函数都可以作为构造函数，只要一个函数是通过 new 来调用的，那么就可以把它称为构造函数。执行构造函数首先会创建一个对象，然后将对象的原型指向构造函数的 prototype 属性，然后将执行上下文中的 this 指向这个对象，最后再执行整个函数，如果返回值不是对象，则返回新建的对象。因为 this 的值指向了新建的对象，因此可以使用 this 给对象赋值。构造函数模式相对于工厂模式的优点是，所创建的对象和构造函数建立起了联系，因此可以通过原型来识别对象的类型。但是构造函数存在一个缺点就是，造成了不必要的函数对象的创建，因为在 js 中函数也是一个对象，因此如果对象属性中包含函数的话，那么每次都会新建一个函数对象，浪费了不必要的内存空间，因为函数是所有的实例都可以通用的。

(3) 第三种模式是原型模式，因为每一个函数都有一个 prototype 属性，这个属性是一个对象，它包含了通过构造函数创建的所有实例都能共享的属性和方法。因此可以使用原型对象来添加公用属性和方法，从而实现代码的复用。这种方式相对于构造函数模式来说，解决了函数对象的复用问题。但是这种模式也存在一些问题，一个是没有办法通过传入参数来初始化值，另一个是如果存在一个引用类型如 Array 这样的值，那么所有的实例将共享一个对象，一个实例对引用类型值的改变会影响所有的实例。

(4) 第四种模式是组合使用构造函数模式和原型模式，这是创建自定义类型的最常见方式。因为构造函数模式和原型模式分开使用都存在一些问题，因此可以组合使用这两种模式，通过构造函数来初始化对象的属性，通过原型对象来实现函数方法的复用。这种方法很好的解决了两种模式单独使用时的缺点，但是有一点不足的就是，因为使用了两种不同的模式，所以对于代码的封装性不够好。

(5) 第五种模式是动态原型模式，这一种模式将原型方法赋值的创建过程移动到了构造函数的内部，通过对属性是否存在的判断，可以实现仅在第一次调用函数时对原型对象赋值一次的效果。这一种方式很好地对上面的混合模式进行了封装。

(6) 第六种模式是寄生构造函数模式，这一种模式和工厂模式的实现基本相同，我对这个模式的理解是，它主要是基于一个已有的类型，在实例化时对实例化的对象进行扩展。这样既不用修改原来的构造函数，也达到了扩展对象的目的。它的一个缺点和工厂模式一样，无法实现对象的识别。

## 2. 对象继承的方式有哪些？

(1) 第一种是以原型链的方式来实现继承，但是这种实现方式存在的缺点是，在包含有引用类型的数据时，会被所有的实例对象所共享，容易造成修改的混乱。还有就是在创建子类型的时候不能向超类型传递参数。

(2) 第二种方式是使用借用构造函数的方式，这种方式是通过在子类型的函数中调用超类型的构造函数来实现的，这一种方法解决了不能向超类型传递参数的缺点，但是它存在的一个问题就是无法实现函数方法的复用，并且超类型原型定义的方法子类型也没有办法访问到。

(3) 第三种方式是组合继承，组合继承是将原型链和借用构造函数组合起来使用的一种方式。通过借用构造函数的方式来实现类型的属性的继承，通过将子类型的原型设置为超类型的实例来实现方法的继承。这种方式解决了上面的两种模式单独使用时的的问题，但是由于我们是以超类型的实例来作为子类型的原型，所以调用了两次超类的构造函数，造成了子类型的原型中多了很多不必要的属性。

(4) 第四种方式是原型式继承，原型式继承的主要思路就是基于已有的对象来创建新的对象，实现的原理是，向函数中传入一个对象，然后返回一个以这个对象为原型的对象。这种继承的思路主要不是为了实现创造一种新的类型，只是对某个对象实现一种简单继承，ES5 中定义的 `Object.create()` 方法就是原型式继承的实现。缺点与原型链方式相同。

(5) 第五种方式是寄生式继承，寄生式继承的思路是创建一个用于封装继承过程的函数，通过传入一个对象，然后复制一个对象的副本，然后对象进行扩展，最后返回这个对象。这个扩展

的过程就可以理解是一种继承。这种继承的优点就是对一个简单对象实现继承，如果这个对象不是自定义类型时。缺点是没有办法实现函数的复用。

(6) 第六种方式是寄生式组合继承，组合继承的缺点就是使用超类型的实例做为子类型的原型，导致添加了不必要的原型属性。寄生式组合继承的方式是使用超类型的原型的副本来作为子类型的原型，这样就避免了创建不必要的属性。

## 九、垃圾回收与内存泄漏

---

### 1. 浏览器的垃圾回收机制

#### (1) 垃圾回收的概念

**垃圾回收：**JavaScript代码运行时，需要分配内存空间来储存变量和值。当变量不在参与运行时，就需要系统收回被占用的内存空间，这就是垃圾回收。

**回收机制：**

- Javascript 具有自动垃圾回收机制，会定期对那些不再使用的变量、对象所占用的内存进行释放，原理就是找到不再使用的变量，然后释放掉其占用的内存。
- JavaScript中存在两种变量：局部变量和全局变量。全局变量的生命周期会持续要页面卸载；而局部变量声明在函数中，它的生命周期从函数执行开始，直到函数执行结束，在这个过程中，局部变量会在堆或栈中存储它们的值，当函数执行结束后，这些局部变量不再被使用，它们所占有的空间就会被释放。
- 不过，当局部变量被外部函数使用时，其中一种情况就是闭包，在函数执行结束后，函数外部的变量依然指向函数内部的局部变量，此时局部变量依然在被使用，所以不会回收。

#### (2) 垃圾回收的方式

浏览器通常使用的垃圾回收方法有两种：标记清除，引用计数。 **1) 标记清除**

- 标记清除是浏览器常见的垃圾回收方式，当变量进入执行环境时，就标记这个变量“进入环境”，被标记为“进入环境”的变量是不能被回收的，因为他们正在被使用。当变量离开环境时，就会被标记为“离开环境”，被标记为“离开环境”的变量会被内存释放。
- 垃圾收集器在运行的时候会给存储在内存中的所有变量都加上标记。然后，它会去掉环境中的变量以及被环境中的变量引用的标记。而在此之后再被加上标记的变量将被视为准备删除

的变量，原因是环境中的变量已经无法访问到这些变量了。最后。垃圾收集器完成内存清除工作，销毁那些带标记的值，并回收他们所占用的内存空间。

## 2) 引用计数

- 另外一种垃圾回收机制就是引用计数，这个用的相对较少。引用计数就是跟踪记录每个值被引用的次数。当声明了一个变量并将一个引用类型赋值给该变量时，则这个值的引用次数就是1。相反，如果包含对这个值引用的变量又取得了另外一个值，则这个值的引用次数就减1。当这个引用次数变为0时，说明这个变量已经没有价值，因此，在在机回收期下次再运行时，这个变量所占有的内存空间就会被释放出来。
- 这种方法会引起**循环引用**的问题：例如：`obj1` 和 `obj2` 通过属性进行相互引用，两个对象的引用次数都是2。当使用循环计数时，由于函数执行完后，两个对象都离开作用域，函数执行结束，`obj1` 和 `obj2` 还将会继续存在，因此它们的引用次数永远不会是0，就会引起循环引用。

javascript 复制代码

```
function fun() {  
    let obj1 = {};  
    let obj2 = {};  
    obj1.a = obj2; // obj1 引用 obj2  
    obj2.a = obj1; // obj2 引用 obj1  
}
```

这种情况下，就要手动释放变量占用的内存：

javascript 复制代码

```
obj1.a = null  
obj2.a = null
```

## (3) 减少垃圾回收

虽然浏览器可以进行垃圾自动回收，但是当代码比较复杂时，垃圾回收所带来的代价比较大，所以应该尽量减少垃圾回收。

- **对数组进行优化：**在清空一个数组时，最简单的方法就是给其赋值为`[]`，但是与此同时会创建一个新的空对象，可以将数组的长度设置为0，以此来达到清空数组的目的。
- **对 `object` 进行优化：**对象尽量复用，对于不再使用的对象，就将其设置为`null`，尽快被回收。
- **对函数进行优化：**在循环中的函数表达式，如果可以复用，尽量放在函数的外面。

## 2. 哪些情况会导致内存泄漏

以下四种情况会造成内存的泄漏：

- **意外的全局变量：** 由于使用未声明的变量，而意外的创建了一个全局变量，而使这个变量一直留在内存中无法被回收。
- **被遗忘的计时器或回调函数：** 设置了 `setInterval` 定时器，而忘记取消它，如果循环函数有对外部变量的引用的话，那么这个变量会被一直留在内存中，而无法被回收。
- **脱离 DOM 的引用：** 获取一个 DOM 元素的引用，而后面这个元素被删除，由于一直保留了对这个元素的引用，所以它也无法被回收。
- **闭包：** 不合理的使用闭包，从而导致某些变量一直被留在内存当中。