

面试

重点：

1、优势：

1. 指定回调函数的方式更加灵活：

旧的：必须在启动异步任务前指定

promise：启动异步任务 \Rightarrow 返回promise对象 \Rightarrow 给promise对象绑定回调函数（甚至可以在异步任务结束后指定）

2. 支持链式调用，可以解决回调地狱问题

(1)什么是回调地狱：

回调函数嵌套调用，外部回调函数异步执行的结果是嵌套的回调函数执行的条件

(2)回调地狱的弊病：

代码不便于阅读、不便于异常的处理

(3)一个不是很优秀的解决方案：

then的链式调用

(4)终极解决方案：

async/await (底层实际上依然使用then的链式调用)

2、catch

catch() 方法返回一个Promise (en-US)，并且处理拒绝的情况。

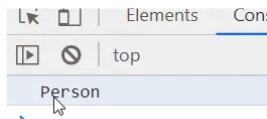
它的行为与调用Promise.prototype.then(undefined, onRejected) 相同。放在函数原型对象上是给实例对象用的 放在自己身上是给自己用的

1、函数对象与实例对象

每一个函数对象有一个不可修改的属性name name的值是函数的名字

```
//函数对象
function Person (){

}
Person.name = 'tom'
console.log(Person.name);
```



new出来的对象为实例对象

2、回调函数分类

回调函数: 我们定义的 我们没有调用 最终执行了

定时器执行 是将回调函数推入了浏览器的回调队列

回调队列里面的东西是等主线程执行完之后执行

```
setTimeout(()=>{
  console.log('@');
},0)
console.log('#');
```

同步回调和主线程平级的

```
//演示同步的回调函数
let arr = [1,3,5,7,9]
arr.forEach((item)=>{
  console.log(item);
})
console.log('主线程的代码');
```

5
7
9
主线程的代码

同步的回调函数

理解: 立即在 **主线程** 上执行, 不会放入回调队列中

例子: 数组遍历相关的回调函数、Promise的executor函数

异步的回调函数

理解: 不会立即执行 会放入 **回调队列**, 等主线程执行完毕后, 按顺序执行

例子: 定时器回调、ajax回调、Promise的成功、失败的回调

3、错误类型的说明

错误类型

· Error: 所有错误的类型

 ReferenceError: 引用的变量不存在

 TypeError: 数据类型不存在

 RangeError: 数据值不在其所允许的范围内 --- 死循环

 SyntaxError: 语法错误

错误处理：

捕获错误: try{}catch(){}

抛出错误: throw error

错误对象

message属性: 错误相关信息

stack属性: 记录信息

js引擎在执行脚本时，发现错误，便会抛出错误

message & stack



4、初始Promise

promise 是什么？

抽象表达：

1、promise是一种新的技术

2、promise是Js异步编程的新方案（旧方案？ --- 纯回调）

具体表达

1、promise是一个内置的构造函数

2、功能上讲，promise的实例对象可以用来封装一个异步操作，并可以获取其成功、失败的值

梳理

1. promise 不是回调，是一个内置的构造函数，是程序员自己new调用的
2. new Promise (executor) 的时候，需要传入一个回调函数 他是一个 **同步的回调**，会立即在主线程上执行它被称为executor函数
3. 每一个promise有三种状态 pending (初始化) fulfilled (成功) rejected (失败)
4. 每一个promise 在被new出来的那一刻，状态都是初始化 (pending)
5. executor函数会接收两个参数，他们都是函数，用形参 resolve 、reject接收
 - (1) 调用resole会使得promise的状态变为fulfilled，同时可以指定成功的value
 - (2) 调用reject会使得promise的状态变为rejected，同时可以指定失败的reason

```
<script type="text/javascript" >
  const p = new Promise((resole,reject)=>{
    //函数体
    console.log("#")
  })
  console.log("@",p)
</script>
```

```
#  
@ ▼ Promise {<pending>} ⓘ  
  ► [[Prototype]]: Promise  
  [[PromiseState]]: "pending"  
  [[PromiseResult]]: undefined
```

```
const myFirstPromise = new Promise((resolve, reject) => {
  //做一些异步操作，最终会调用下面两者之一：
  //    resolve(someValue); // fulfilled
  //或
  //    reject("failure reason"); // rejected
})
```

5、then

`then()` 方法返回一个 `Promise`。

它最多需要有两个参数：`Promise` 的成功和失败情况的回调函数

通过 `then` 方法为 `Promise` 的实例指定成功、失败的回调函数，来获取成功的 `value` 失败的 `reason`

注意：`then` 方法所指定的：成功的回调、失败的回调 都是**异步的回调**

因为 `Promise.prototype.then` 和 `Promise.prototype.catch` 方法返回的是 `promise`，所以它们可以被链式调用

注意：如果忽略针对某个状态的回调函数参数，或者提供非函数（`nonfunction`）参数，那么 `then` 方法将会丢失关于该状态的回调函数信息，但是并不会产生错误。

如果调用 `then` 的 `Promise` 的状态（`fulfillment` 或 `rejection`）发生改变，但是 `then` 中并没有关于这种状态的回调函数，那么 `then` 将创建一个没有经过回调函数处理的新 `Promise` 对象，这个新 `Promise` 只是简单地接受调用这个 `then` 的原 `Promise` 的终态作为它的终态。

关于状态的注意点：

1.三个状态：

`pending`: 未确定的-----初始状态

`fulfilled`: 成功的-----调用`resolve()`后的状态

`rejected`: 失败的-----调用`reject()`后的状态

2.两种状态改变

`pending` \Rightarrow `fulfilled`

`pending` \Rightarrow `rejected`

3.状态只能改变一次！！（`then`里面只有一个方法会被调用）

4.一个`promise`指定多个成功/失败回调函数，都会调用吗？**都会**，底层使用的是队列！

```

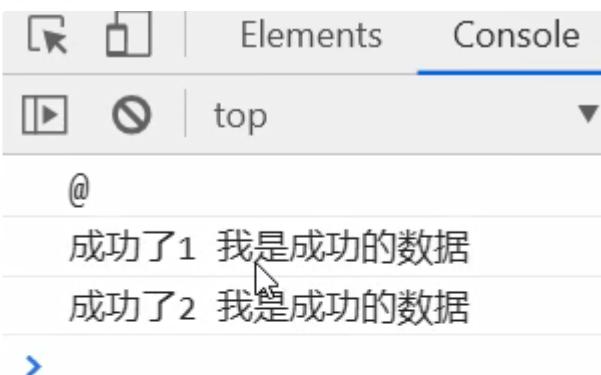
<-->
<script type="text/javascript" >

  const p = new Promise((resolve,reject)=>{
    setTimeout(()=>{
      resolve('我是成功的数据')
    },2000)
  })

  p.then(
    (value)=>{console.log('成功了1',value);}, //成功的回调-异步
    (reason)=>{console.log('失败了1',reason);} //失败的回调-异步
  )
  p.then(
    (value)=>{console.log('成功了2',value);}, //成功的回调-异步
    (reason)=>{console.log('失败了2',reason);} //失败的回调-异步
  )
  console.log('@');

</script>

```



6、使用promise 和 xhr 封装ajax

```

function sendAjax(url, data) {
  return new Promise((resolve, reject) => {
    //实例xhr
    const xhr = new XMLHttpRequest();
    //绑定监听
    xhr.onreadystatechange = () => {
      if (xhr.readyState === 4) {
        if (xhr.status ≥ 200 && xhr.status < 300)
          resolve(xhr.response);
        else reject("请求出了点问题");
      }
    }
  })
}

```

```

    }
};

//整理参数
let str = "";
for (let key in data) {
    str += `${key}=${data[key]}&`;
}
str = str.slice(0, -1);
xhr.open("GET", url + "?" + str);
xhr.responseType = "json";
xhr.send();
});

}

const x = sendAjax("https://api.apiopen.top/getJoke", {
    page: 1,
    count: 2,
    type: "video",
});

x.then(
    (data) => {
        console.log("成功了", data);
    },
    (reason) => {
        console.log("失败了", reason);
    }
);

```

这一段不在主线程上执行

```

1 function sendAjax(url,data){
2     return new Promise((resolve,reject)=>{
3         //实例xhr
4         const xhr = new XMLHttpRequest()
5         //绑定监听
6         xhr.onreadystatechange = ()=>{
7             if(xhr.readyState === 4){
8                 if(xhr.status >= 200 && xhr.status < 300) resolve(xhr.response);
9                 else reject('请求出了点问题');
10            }
11        }
12        //整理参数
13        let str = ''
14        for (let key in data){

```

7、纯回调封装ajax

自定义success成功回调

自定义error失败回调

手动设置：xhr.status为2xx的时候调用success 其他调用error

```
//  
function sendAjax(url,data,success,error){  
    //实例xhr  
    const xhr = new XMLHttpRequest()  
    //绑定监听  
    xhr.onreadystatechange = ()=>{ ...  
        //整理参数  
        let str = ''  
        for (let key in data){ ...  
            str = str.slice(0,-1)  
            xhr.open('GET',url+'?'+str)  
            xhr.responseType = 'json'  
            xhr.send()  
    }  
  
    sendAjax(  
        'https://api.apiopen.top/getJoke',  
        {page:1,count:2,type:'video'},  
        (response)=>{console.log('成功了',response)}, //成功的回调  
        (err)=>{console.log('失败了',err)} //失败的回调  
    )  
}
```

8、包管理器的对比

cnpm 下载包的时候 会多一个快捷方式

移除包的时候会移除所有

npm 当自己的包名 (npm init的时候) 和主流的包名重复的时候

不能安装主流包名

yarn 没有上述问题

包管理器的使用：使用yarn的命令 cnpm的仓库地址

9、promise 的API

1. Promise构造函数: new Promise (executor) {}

executor函数: 是同步执行的, (resolve, reject) => {}

resolve函数: 调用resolve将Promise实例内部状态改为成功(fulfilled)。

reject函数: 调用reject将Promise实例内部状态改为失败(rejected)。

说明: executor函数会在Promise内部立即同步调用, 异步代码放在executor函数中。

2. Promise.prototype.then方法: Promise实例.then(onFulfilled, onRejected)

onFulfilled: 成功的回调函数 (value) => {}

onRejected: 失败的回调函数 (reason) => {}

特别注意(难点): then方法会返回一个新的Promise实例对象

3. Promise.prototype.catch方法: Promise实例.catch(onRejected)

onRejected: 失败的回调函数 (reason) => {}

说明: catch方法是then方法的语法糖, 相当于: then(undefined, onRejected)

4. Promise.resolve方法: Promise.resolve(value)

说明: 用于快速返回一个状态为fulfilled或rejected的Promise实例对象

备注: value的值可能是: (1)非Promise值 (2)Promise值

5. Promise.reject方法: Promise.reject方法(reason)

说明: 用于快速返回一个状态必为rejected的Promise实例对象

6. Promise.all方法: Promise.all(promiseArr)

promiseArr: 包含n个Promise实例的数组

说明: 返回一个新的Promise实例, 只有所有的promise都成功才成功, 只要有一个失败了就直接失败。

7. Promise.race方法: Promise.race(promiseArr)

promiseArr: 包含n个Promise实例的数组

说明: 返回一个新的Promise实例, 成功还是很失败? 以最先出结果的promise为准。

(1) catch()

catch() 方法返回一个Promise (en-US)，并且处理拒绝的情况。它的行为与调用 Promise.prototype.then(undefined, onRejected) 相同

注意：promise失败，却未指定失败的回调 会报错

但是成功了 没有指定成功的回调 却不会报错

```
@ ▶ Promise {<rejected>: "ok"}  
✖ ▶ Uncaught (in promise) ok  
>
```

阿里外包面试题

```
//Promise.prototype.catch方法  
const p = new Promise((resolve,reject)=>{  
    setTimeout(()=>{  
        reject(100)  
    },1000)  
})  
  
p.then(  
    value => {console.log('成功了',value);}  
)  
  
p.catch(  
    reason => {console.log('失败了',reason);}  
)
```

队列



(2) resolve() & reject()

Promise.resolve(value)

说明：用于快速返回一个状态为fulfilled或rejected的Promise实例对象

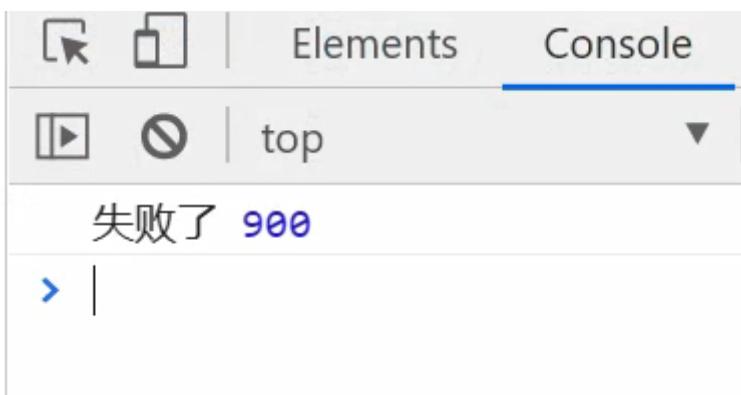
备注: value的值可能是: (1)非Promise值 (成功的结果) (2)Promise值, 由传入的 promise状态决定

Promise.reject方法: Promise.reject方法(reason)

说明: 用于快速返回一个状态**必为rejected**的Promise实例对象

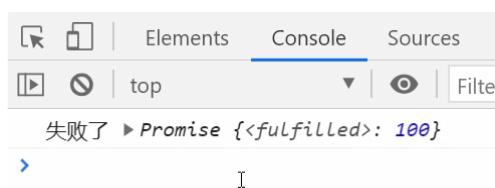
reject 传递一个失败的promise 实例

```
//Promise.resolve
const p0 = Promise.reject(900)
const p = Promise.resolve(p0)
p.then(
  value => {console.log('成功了',value);},
  reason => {console.log('失败了',reason);}
)
```



reject 传递一个成功的promise 实例

```
//Promise.reject
const p0 = Promise.resolve(100)
const p = Promise.reject(p0)
p.then([
  value => {console.log('成功了',value);},
  reason => {console.log('失败了',reason);}
])
```



(3) all()

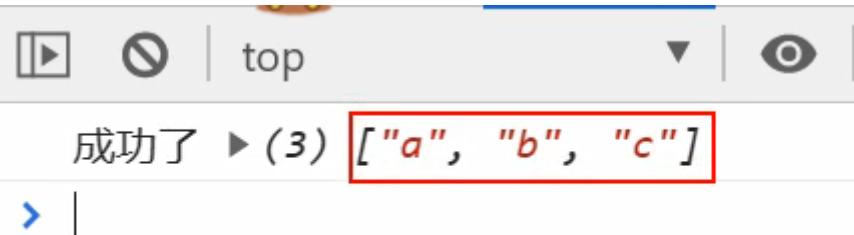
返回一个新的Promise实例, 只有所有的promise都成功才成功, 只要有一个失败了就直接失败。和传入的顺序无关

成功的返回结果：所有成功的结果组成的数组

失败的返回结果：谁失败，返回谁的失败结果

```
//Promise.all
const p1 = Promise.resolve('a')
const p2 = new Promise((resolve,reject)=>{
  setTimeout(()=>{
    resolve('b')
  },500)
})
const p3 = new Promise((resolve,reject)=>{
  setTimeout(()=>{
    resolve('c')
  },2000)
})
```

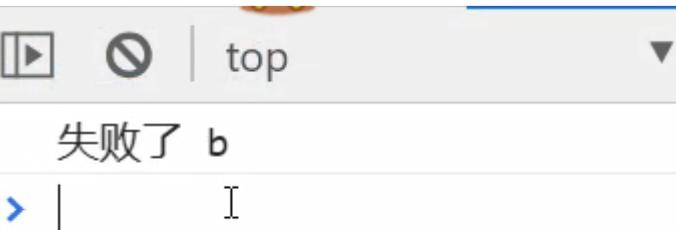
```
const x = Promise.all([p1,p2,p3])
x.then(
  value => {console.log('成功了',value);},
  reason => {console.log('失败了',reason);}
)
</script>
```



```
//Promise.all
const p1 = Promise.resolve('a')
const p2 = new Promise((resolve,reject)=>{
  setTimeout(()=>{
    reject('b')
  },500)
})
const p3 = new Promise((resolve,reject)=>{
  setTimeout(()=>{
    resolve('c')
  },2000)
})
```

b已经失败了 不等c了

```
const x = Promise.all([p1,p2,p3])
x.then(
  value => {console.log('成功了',value);},
  reason => {console.log('失败了',reason);}
)
</script>
```



(4) race()

传你个promise组成的数组

返回一个新的Promise实例，成功还是很失败？以最先出结果的promise为准。

不用等 第一个成功就是成功的！和传入的顺序无关

```
//Promise.race
const p1 = Promise.resolve('a')  
const p2 = new Promise((resolve,reject)=>{  
    setTimeout(()=>{  
        reject('b')  
    },500)  
})  
const p3 = new Promise((resolve,reject)=>{  
    setTimeout(()=>{  
        reject('c')  
    },2000)  
})  
const x = Promise.race([p1,p2,p3])  
x.then(  
    value => {console.log('成功了',value);},  
    reason => {console.log('失败了',reason);}  
)  
</script>
```

成功了 a

>

9、如何改变promise实例的状态

如何改变一个Promise实例的状态？

- (1) 执行`resolve(value)`: 如果当前是pending就会变为fulfilled
- (2) 执行`reject(reason)`: 如果当前是pending就会变为rejected
- (3) 执行器函数`(executor)`抛出异常: 如果当前是pending就会变为rejected

状态只能改变一次！

```
const p = new Promise((resolve,reject)=>{
    console.log(a); //引擎抛异常
    // throw 900 //编码抛异常
})
p.then(
    value => {console.log('成功了',value);},
    reason => {console.log('失败了',reason);}
)
```

失败了 ReferenceError: a is not defined
at 08_promise的几个关键问题1.html:16
at new Promise (<anonymous>)
at 08_promise的几个关键问题1.html:15

```
<script type="text/javascript" >
const p = new Promise((resolve,reject)=>{
    resolve(100)
    console.log(a);
})
p.then(
    value => {console.log('成功了',value);},
    reason => {console.log('失败了',reason);}
)
</script>
```

成功了 100

10、改变状态与指定回调（then）的顺序

改变promise实例的状态和指定回调函数的状态谁先谁后？

1. 都有可能，正常先指定回调再改变状态，但也可以先改状态再指定回调

2. 如何先改状态在指定回调？

延迟一会在调用then

3. promise实例什么时候才能得到数据

先指定回调，状态改变，回调就会调用，得到数据

先改变状态，指定回调时，回调就会调用，得到数据

//先改状态，后指定回调

```
const p = new Promise((resolve,reject)=>{
    resolve('a')
})
setTimeout(()=>{
    p.then(
        value => {console.log('成功了',value);},
        reason => {console.log('失败了',reason);}
    )
},3000)
```

11、then的链式调用

(1) 值和状态由什么决定

then()返回的是一个新的Promise实例，它的值和状态由什么决定？

1、简单表达：由then指定的回调函数的结果决定

2、复杂表达：如果then指定的回调函数返回的结果是

(1)如果then所指定的回调返回的是非Promise值a：

那么【新Promise实例】状态为：成功(fulfilled)，成功的value为a

(2)如果then所指定的回调抛出异常：

那么【新Promise实例】状态为失败(rejected)，reason为抛出的那个异常

(3)如果then所指定的回调返回的是一个Promise实例p：

那么【新Promise实例】的状态、值，都与p一致

3. 中断promise链

(1) .当使用promise的链式调用时，在中间中断，不再调用后面的回调函数

(2) .办法：在失败的回调函数中返回一个pending状态的Promise实例

由着2.3可知，返回一个全新的promise实例，则既不会走成功的也不会走失败的，因此中断

```
<script>
const p = new Promise((resolve,reject)=>{
  setTimeout(()=>{
    | reject('a')
  },1000)
})

const x = p.then(
  value => {console.log('成功了1',value); return Promise.reject('a')},
  reason => {console.log('失败了1',reason);}
)

x.then(
  value => {console.log('成功了2',value)},
  reason => {console.log('失败了2',reason);}
)
</script>
</body>
```

```
<script>
const p = new Promise((resolve,reject)=>{
  setTimeout(()=>{
    | resolve('a')
  },1000)
})

p.then(
  value => {console.log('成功 1',value); return Promise.reject('a')},
  reason => {console.log('失败了 1',reason);}
).then(
  value => {console.log('成功了2',value);return true},
  reason => {console.log('失败了2',reason); return 100}
).then(
  value => {console.log('成功了3',value);return false},
  reason => {console.log('失败了3',reason); return false}
).then(
  value => {console.log('成功了4',value);return -100},
  reason => {console.log('失败了4',reason);}
)
</script>
</body>
```

The screenshot shows the browser's developer tools console with two separate promise chains. The top chain starts with a rejection ('失败了1 a'), followed by a success ('成功了2 undefined'), and then another rejection ('失败了2 undefined'). The bottom chain starts with a success ('成功 1 a'), followed by three rejections ('失败了 1 a', '失败了2 a', and '失败了3 a'). Red boxes highlight specific code segments in the script, and arrows show the flow from one part to the next in each chain.

(2) 中断 promise 链

当使用promise的then链式调用时，在中间中断，不再调用后面的回调函数。

在失败的回调函数中返回一个 **pending** 状态的Promise实例。

```

function sendAjax(url,data){ ...
    //发送第1次请求
    sendAjax('https://api.apiopen.top/getJoke',{page:1})
    .then(
        value => {
            console.log('第1次请求成功了',value);
            //发送第2次请求
            return sendAjax('https://api.apiopen.top/getJoke2',{page:1})
        },
        reason => {console.log('第1次请求失败了',reason);return new Promise(()=>{})}
    )
    .then(
        value => {
            console.log('第2次请求成功了',value);
            //发送第3次请求
            return sendAjax('https://api.apiopen.top/getJoke',{page:1})
        },
        reason => {console.log('第2次请求失败了',reason);return new Promise(()=>{})}
    )
    .then(
        value => {console.log('第3次请求成功了',value)},
        reason => {console.log('第3次请求失败了',reason)}
    )
}

```

404异常
至此中断

(3) 错误的穿透

promise错误穿透：

(1) 当使用promise的then链式调用时，**不指定失败的回调**的情况下，可以在最后用catch指定一个失败的回调，

(2) 前面任何操作出了错误，都会传到最后失败的回调中处理了

备注：如果不存在then的链式调用，就不需要考虑then的错误穿透。

产生的问题：

```

//另一个例子演示错误的穿透
const p = new Promise((resolve,reject)=>{
  setTimeout(()=>{
    | reject(-1)
  },1000)
})

p.then(
  value => {console.log('成功了1',value);return 'b'},
  reason => {console.log('失败了1',reason);return f2}
)
.then(
  value => {console.log('成功了2',value);return 'c'},
  reason => {console.log('失败了2',reason);return -3}
)

```



错误的穿透

不指定失败的回调的情况下，**使用catch兜底**

```

//发送第1次请求
sendAjax('https://api.apioopen.top/getJoke',{page:1})
.then(
  value => {...}
  // reason => {console.log('第1次请求失败了',reason);return new Promise(()=>{})}
)
.then(
  value => {...}
  // reason => {console.log('第2次请求失败了',reason);return new Promise(()=>{})}
)
.then(
  value => {console.log('第3次请求成功了',value)},
  // reason => {console.log('第3次请求失败了',reason);return new Promise(()=>{})}
)
.catch(
  reason => console.log(reason)
)

```

底层原理

给每个失败的回调一个抛出异常，那么后面就会一直走失败的回调，直到最后被catch

```

//另一个例子演示错误的穿透
const p = new Promise((resolve,reject)=>{
  setTimeout(()=>{
    reject(-1)
  },1000)
})

p.then(
  value => {console.log('成功了1',value);return 'b'},
  reason => {throw reason}
)
.then(
  value => {console.log('成功了2',value);return Promise.reject(-108)},
  reason => {throw reason}
)
.catch(
  reason => {console.log('失败了',reason)}
)

```

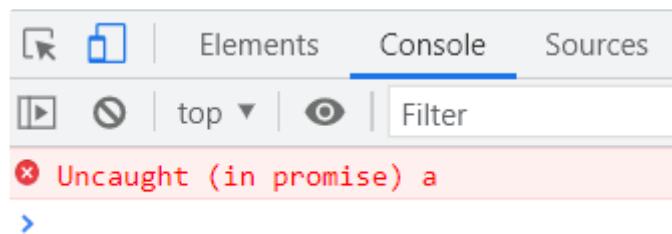
12、await 和 async

(1) await和async的使用

注意

- 1、await右侧的表达式一般为Promise实例对象，但也可以是其它的值
- 2、await只能等到成功的结果，失败的结果会报错，未被抓住的错误

注意：a是返回的错误 `reject("a")`



- 3、await必须被包裹在async修饰的方法的里面

笔记

1. async修饰的函数

`async`写在`function`关键字前面

函数的返回值为 `promise`实例对象

`Promise`实例的结果由`async`函数执行的返回值决定

2. await表达式

await右侧的表达式一般为Promise实例对象，但 **也可以是其它的值**

(1).如果表达式是Promise实例对象，将等待，await后的返回值是promise成功的值

(2).如果表达式是其它值，直接将此值作为await的返回值，不需要等待

```
let a = await 100;
```

3. 注意：

await必须写在async函数中，但async函数中可以没有await

如果await的Promise实例对象失败了，就会抛出异常，需要通过try...catch来捕获处理

(2)原理

await的原理是 最终还是使用了.then 把函数中的其他代码放在了成功的回调里面

若我们使用async配合await这种写法：

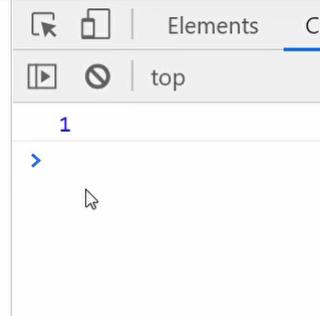
- 1.表面上不出现任何的回调函数
- 2.但实际上底层把我们写的代码进行了加工，把回调函数“还原”回来了。
- 3.最终运行的代码是依然有回调的，只是程序员没有看见。

示例：

1 一开始就展示了

```
<script>
  const p = new Promise((resolve,reject)=>{
    setTimeout(()=>{
      |  resolve('a')
    },4000)
  })

  async function demo(){
    const result = await p
    console.log(result);
  }
  demo()
  console.log(1);
```



2、await的原理是 最终还是使用了.then 把函数中的其他代码放在了成功的回调里面

```

const p = new Promise((resolve,reject)=>{
  setTimeout(()=>{
    | resolve('a')
  },4000)
})

async function demo(){
  p.then(
    result => {
      console.log(result);
      console.log(100);
      console.log(200);
    },
  )
}

demo()

```

13、宏队列与微队列

注意：

异步的回调不能直接在主线程上执行 而是加入到回调队列排队，待主线程执行完毕后再按顺序执行

- 1、回调队列分宏队列和微队列，
- 2、微队列优先级高于宏队列 先执行微队列， **每次要执行宏队列里的一个任务** 之前，先看微队列里是否有待执行的微任务
- 3、常见的微任务：promise的回调

笔记：

1. JS中用来存储待执行回调函数的队列包含2个不同特定的队列
 1. 宏队列：用来保存待执行的宏任务（回调），比如：定时器回调/DOM事件回调/ajax回调
 2. 微队列：用来保存待执行的微任务（回调），比如：promise的回调/MutationObserver的回调

宏队列：[宏任务1, 宏任务2.....]

微队列：[微任务1, 微任务2.....]

规则： **每次要执行宏队列里的一个任务** 之前，先看微队列里是否有待执行的微任务

- 1.如果有，先执行微任务
- 2.如果没有，按照宏队列里任务的顺序，依次执行

```
setTimeout(()=>{
  console.log('timeout')
},0)

Promise.resolve(1).then(
  value => console.log('成功1',value)
)
Promise.resolve(2).then(
  value => console.log('成功2',value)
)
console.log('主线程')
```

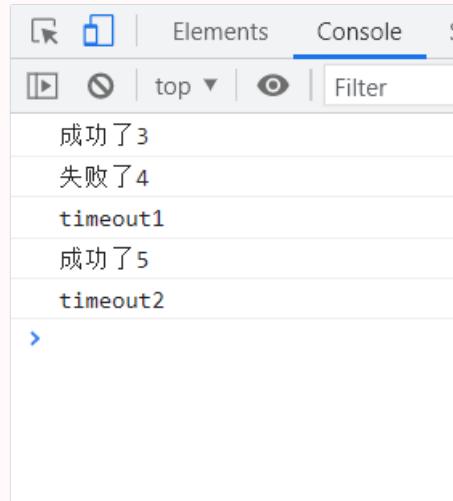
```
主线程
成功1 1
成功2 2
timeout
> |
```

(1)面试题

注意： 每次要执行宏队列里的一个任务 之前，先看微队列里是否有待执行的微任务

```
//代码三
setTimeout(()=>{
  console.log('timeout1')
  Promise.resolve(5).then(
    value => console.log('成功了5')
  )
}
setTimeout(()=>{
  console.log('timeout2')
}

Promise.resolve(3).then(
  value => console.log('成功了3')
)
Promise.resolve(4).then(
  value => console.log('成功了4')
)
```



(2)面试题

- 1、什么时候进队列？ 需要执行回调的时候
- 2、promise先指定then里面的回调，回调没有丢，而是缓存在实例对象身上，执行的时候仍然要推进微队列
- 3、new Promise (executor) 传入的执行器函数是一个同步的回调 在主线程执行

```
setTimeout(() => {
  console.log("0")
},0)
new Promise((resolve, reject) => {
  console.log("1")
  resolve()
}).then(() => {
  console.log("2")
  new Promise((resolve, reject) => [
    console.log("3")
    resolve()
  ]).then(() => {
    console.log("4")
  }).then(() => {
    console.log("5")
  })
}).then(() => {
  console.log("6")
})
new Promise((resolve, reject) => [
  console.log("7")
  resolve()
]).then(() => {
  console.log("8")
})
```

The screenshot shows the browser's developer tools with the 'Console' tab selected. The output area displays the following sequence of numbers:

```
1
7
2
3
8
4
6
5
0
> |
```

A cursor icon is visible at the bottom right of the console window.