

03-Promise

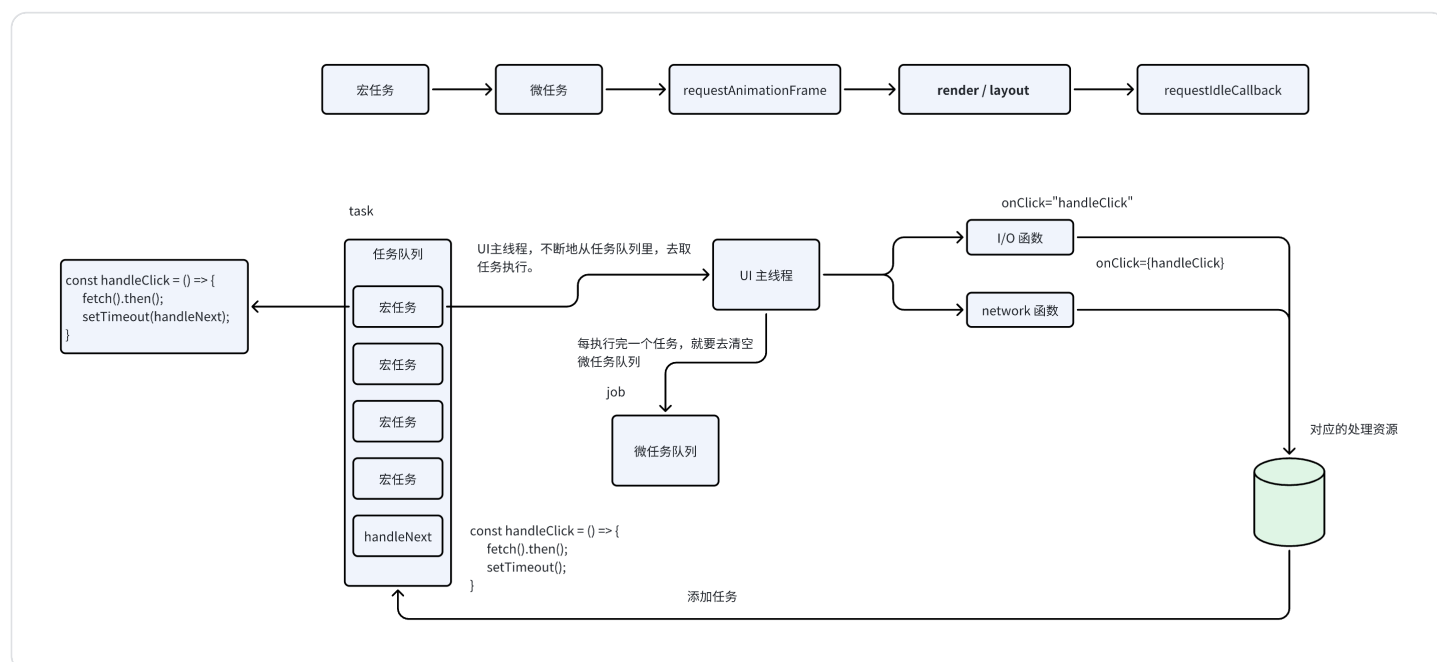
Async / await 简介

Promise A+ 规范

实现 Promise

1. 异步的逻辑

浏览器



为什么会有微任务?

- 主线程执行消息队列的宏任务, 粒度有点不够, 微任务的时效性强;
- 微任务可以改变当前的编程模型;
- 如果数据量大, 可以解决一些异步时机不可控的问题。

异步的发展

```
1 function foo() {  
2   const bar = "bar";  
3 }  
4
```

```

5 // 我如果希望通过调用 foo, 拿到这个 bar。很简单:
6 function foo() {
7     const bar = "bar";
8     return bar;
9 }
10
11 // 我如果希望通过调用 foo 的 1000ms 以后, 再拿到这个 bar, 怎么办?
12 function foo(cb: Function) {
13     const bar = "bar";
14     setTimeout(() => {
15         cb(bar);
16     }, 1000)
17 };
18
19 const handleFoo = (res) => {
20     // res 就是我们想要的 bar;
21     // 但是, 我想基于得到 bar 的值以后, 再写新的逻辑, 只能在这里写了。
22
23 }
24
25 foo(handleFoo);
26
27

```

Callback

当没有 Promise 的时候, 大量的异步逻辑回调, 都依赖于, callback

在 node 中, 大量这种用法

```

1 fs.readFile("a.txt", "utf-8", function(err, data) {
2     fs.readFile("b.txt", "utf-8", function(err, data) {
3         fs.readFile("c.txt", "utf-8", function(err, data) {
4
5             })
6         })
7     })
8

```

Promise

应用 -> fetch, webpack,

```
1 //
2
3 export const getData = () => post('xxxx/xxx');
4
5
6 getData().then(res => {
7     // vue
8     this.dataList = res.data;
9     // react
10    setData(res.data)
11 })
12
13
14 // -----
15
16 function post(url) {
17     return new Promise((resolve, reject) => {
18         setTimeout(() => {
19             resolve({data: [1,2,3]})
20         }, 1000)
21     })
22 }
23
```

Generator

- 协程，是一种比线程更小的机制，但是本质上使用很少，所以一般也不会问。

```
1 function *gen() {
2     yield "1st messian";
3     yield "2nd messian";
4     let res = yield "3rd messian";
5     return res;
6 }
7
8 let result = gen();
9
10 console.log(result.next());
11 console.log(result.next());
12 console.log(result.next());
```

```
13 console.log(result.next("over"));
```

async / await

异步编程的一种重大改进，提供了一种在不阻塞主线程的情况下，使用同步代码直接执行异步的逻辑。

```
1
2 // -----
3
4 async function post(url) {
5     return new Promise((resolve, reject) => {
6         setTimeout(() => {
7             resolve({data: [1,2,3], url})
8         }, 1000)
9     })
10 }
11
12 const getData = async () => await post('xxxx/xxx');
13
14 const run = async () => {
15     console.log('starting...')
16     const res = await getData();
17     console.log(res)
18 };
19
20 run();
21
```

Promise 深入理解

初探 promise

```
1 export const getData = () => post('xxxx/xxx');
2 getData().then(res => {
3     // vue
4     this.dataList = res.data;
5     // react
```

```

6     setData(res.data)
7 }, err => {
8
9 })
10
11 function post(url) {
12     return new Promise((resolve, reject) => {
13         setTimeout(() => {
14             resolve({data: [1,2,3]})
15         }, 1000)
16     })
17 }

```

规则



- Promise 是一个构造函数；
- Promise 接受一个函数作为参数，这个函数的参数，是两个函数(`resolve` , `reject`)
- Promise 返回一个对象，这个对象包含一个 `then` 函数，这个 `then` 函数，接收两个参数，这两个参数，也都是函数。
- Promise 的 `status` :
 - `pending`
 - 初始的状态，可以改变
 - 一个 `Promise` 在 `resolve` 或者 `reject` 之前，都处于这个状态
 - 我们可以通过调用 `resolve` 或者 `reject` 方法，让这个 `Promise` 变成 `fulfilled` 或者 `rejected` 的状态。
 - `fulfilled`
 - 不可变状态
 - 在 `resolve` 之后，变成这个状态，拥有一个 `value`
 - `rejected`
 - 不可变状态
 - 在 `reject` 之后，变成这个状态，拥有一个 `reason`
- `then` 函数
 - 参数
 - `onFulfilled` , `onRejected` 必须是函数类型，如果不是，应该被忽略；

- `onFulfilled` 和 `onRejected` 的特性
 - 在 `promise` 变成 `fulfilled` / `rejected` 状态的时候, 应该调用 `onFulfilled` / `onRejected` ;
 - 在 `promise` 变成 `fulfilled` / `rejected` 状态之前, 不应该被调用
 - 只能调用一次。

实现

```
1
2 function LPromise(execute) {
3   this.status = "pending";
4   this.value = null;
5   this.reason = null;
6
7   const resolve = (value) => {
8     if(this.status === "pending") {
9       this.value = value;
10      this.status = "fulfilled";
11    }
12  }
13
14  const reject = (reason) => {
15    if(this.status === "pending") {
16      this.reason = reason;
17      this.status = "rejected";
18    }
19  }
20
21  execute(resolve, reject);
22 }
23
24 LPromise.prototype.then = function(onFulfilled, onRejected) {
25   onFulfilled = typeof onFulfilled === "function" ? onFulfilled: (data) => { r
26   onRejected = typeof onRejected === "function" ? onRejected: (error) => { thr
27
28   if(this.status === "fulfilled") {
29     onFulfilled(this.value);
30   }
31
32   if(this.status === "rejected") {
33     onRejected(this.reason);
34   }
35 }
```

测试：

```
1 new LPromise((resolve, reject) => {
2   resolve('hello luyi')
3 }).then(res => {
4   console.log(res)
5 })
```

问题：

```
1 new LPromise((resolve, reject) => {
2   setTimeout(() => {
3     resolve('hello luyi');
4   }, 1000)
5 }).then(res => {
6   console.log(res)
7 })
8
9
```

啥也没有了。

因为我们在 resolve 执行的时候，then 函数已经执行过了。

- then 是不是要等到我们 resolve 的时候再执行？？？
- 所以，我们要在一个合适的时机，去执行 then 的 `onFulfilled`
- 发布订阅。

Promise 进阶

规则



- resolve / reject 执行了之后，再执行 onfulfilled 和 onrejected;
- onfulfilled 和 onrejected 应该是微任务。

实现

```
1
2 function LPromise(execute) {
3     this.status = "pending";
4     this.value = null;
5     this.reason = null;
6
7     this.onFulfilledArray = [];
8     this.onRejectedArray = [];
9
10    const resolve = (value) => {
11        queueMicrotask(() => {
12            if(this.status === "pending") {
13                this.value = value;
14                this.status = "fulfilled";
15                this.onFulfilledArray.forEach(func => func(value))
16            }
17        })
18    }
19
20    const reject = (reason) => {
21        queueMicrotask(() => {
22            if(this.status === "pending") {
23                this.reason = reason;
24                this.status = "rejected";
25                this.onRejectedArray.forEach(func => func(reason))
26            }
27        })
28    }
29
30    execute(resolve, reject);
31 }
32
33 LPromise.prototype.then = function(onFulfilled, onRejected) {
34     onFulfilled = typeof onFulfilled === "function" ? onFulfilled: (data) => { r
35     onRejected = typeof onRejected === "function" ? onRejected: (error) => { thr
36
37     if(this.status === "fulfilled") {
38         onFulfilled(this.value);
39     }
40
41     if(this.status === "rejected") {
42         onRejected(this.reason);
43     }
```



```
44
45     if(this.status === "pending") {
46         this.onFulfilledArray.push(onFulfilled);
47         this.onRejectedArray.push(onRejected);
48     }
49 }
50
```

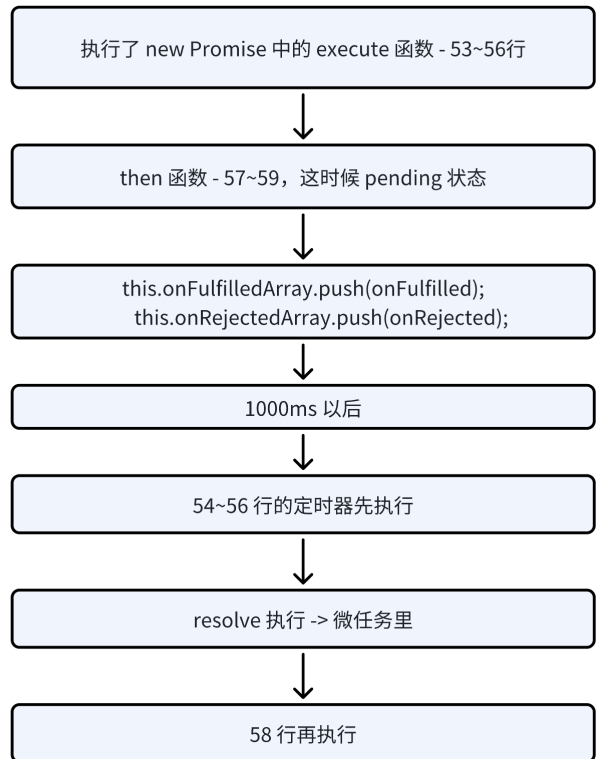
数组 push 的作用和直接拿过来用有啥区别吗???

```
1  const p = new LPromise((resolve, reject) => {
2      setTimeout(() => {
3          resolve('hello luyi');
4          console.log("settimeout")
5      }, 1000)
6  });
7  p.then(res => {
8      console.log(res);
9      return res + "luyi"
10 });
11 p.then(res => {
12     console.log(res+"2");
13     return res + "luyi"
14 })
15
16
```

```

52 new LPromise((resolve, reject) => {
53   setTimeout(() => {
54     resolve('hello luyi');
55     console.log("settimeout")
56   }, 1000)
57 }).then(res => {
58   console.log(res)
59 })

```



问题：

```

1 new LPromise((resolve, reject) => {
2   setTimeout(() => {
3     resolve('hello luyi');
4     console.log("settimeout")
5   }, 1000)
6 }).then(res => {
7   console.log(res); // hello luyi
8   return res + "luyi"
9 }).then(res => {
10    console.log(res) // hello luyi luyi
11  })
12

```

以上不work。

Promise 的链式调用

规则



- then 方法，应该返回一个 Promise

```
promise2 = promise1.then(onFulfilled, onRejected)
```

- onFulfilled / onRejected 的执行结果，为 x，调用 resolvePromise
- 如果 onFulfilled / onRejected 执行时抛出异常，我们 promise2 需要被 reject
- 如果 onFulfilled / onRejected 不是一个函数，promise2 以 promise1 的 value 或者 reason 触发 fulfilled 和 rejected

promise1 中 onfulfilled 返回了一个值，这个值需要被 promise2 进行 resolve，才能出现在下一个 then(res)。

实现

```
1
2 function LPromise(execute) {
3   this.status = "pending";
4   this.value = null;
5   this.reason = null;
6
7   this.onFulfilledArray = [];
8   this.onRejectedArray = [];
9
10  const resolve = (value) => {
11    queueMicrotask(() => {
12      if(this.status === "pending") {
13        this.value = value;
14        this.status = "fulfilled";
15        this.onFulfilledArray.forEach(func => func(value))
16      }
17    })
18  }
19
20  const reject = (reason) => {
21    queueMicrotask(() => {
22      if(this.status === "pending") {
23        this.reason = reason;
24        this.status = "rejected";
25        this.onRejectedArray.forEach(func => func(reason))
26      }
27    })
28  }
29  // try catch
30  execute(resolve, reject);
31 }
32
33 LPromise.prototype.then = function(onFulfilled, onRejected) {
34   onFulfilled = typeof onFulfilled === "function" ? onFulfilled: (data) => { r
```

```

35     onRejected = typeof onRejected === "function" ? onRejected: (error) => { thr
36
37     let promise2;
38
39     if(this.status === "fulfilled") {
40         return promise2 = new LPromise((resolve, reject) => {
41             queueMicrotask(() => {
42                 try {
43                     // promise1 中 onfulfilled 返回了一个值, 这个值需要被 promise2 ;
44                     let result = onFulfilled(this.value);
45                     resolve(result);
46                 } catch(e) {
47                     reject(e)
48                 }
49             })
50         })
51     }
52
53
54     if(this.status === "rejected") {
55         return promise2 = new LPromise((resolve, reject) => {
56             queueMicrotask(() => {
57                 try {
58                     // promise1 中 onfulfilled 返回了一个值, 这个值需要被 promise2 ;
59                     let result = onRejected(this.reason);
60                     resolve(result);
61                 } catch(e) {
62                     reject(e)
63                 }
64             })
65         })
66     }
67
68     if(this.status === "pending") {
69         return promise2 = new LPromise((resolve, reject) => {
70             this.onFulfilledArray.push(() => {
71                 try {
72                     let result = onFulfilled(this.value);
73                     resolve(result);
74                 } catch(e) {
75                     reject(e)
76                 }
77             });
78             this.onRejectedArray.push(() => {
79                 try {
80                     // promise1 中 onfulfilled 返回了一个值, 这个值需要被 promise2 ;
81                     let result = onRejected(this.reason);

```

```

82         resolve(result);
83     } catch(e) {
84         reject(e)
85     }
86     });
87 })
88
89 }
90 }
91
92

```

选读：resolvePromise 规范

规则

`resolvePromise(promise2, x, resolve, reject)`

- 如果 promise2 和 x 相等，那么 reject error;
- 如果 promise2 是一个 promise
 - 如果 x 是一个 pending 状态，那么 promise2 必须要再 pending, 直到 x 变成 fulfilled / rejected
 - 如果 x 被 fulfilled，fulfill promise with the same value
 - 如果 x 被 rejected，reject promise with the same reason
- 如果 x 是一个 object 或者 function
 - Let thenable = x.then
 - 如果 x.then 这一步出错，那么 reject promise with e as the reason
 - 如果 then 是一个函数，then.call(x, resolvePromiseFn, rejectPromiseFn)
 - resolvePromiseFn 的入参是 y, 执行 `resolvePromise(promise2, y, resolve, reject)`
 - rejectPromiseFn 的入参是 r, reject promise with r
 - 如果 resolvePromiseFn 和 rejectPromiseFn 都调用了，那么第一个调用优先，后面的忽略
 - 如果调用 then 抛出异常
 - 如果 resolvePromise 或 rejectPromise 已经被调用，可以忽略
 - 如果 then 不是一个 function，fulfill promise with x

实现

```
1
2 const resolvePromise = (promise2, result, resolve, reject) => {
3   // 当 result 和 promise2 相等时, 也就是说 onfulfilled 返回 promise2 时, 进行 rej
4   if (result === promise2) {
5     reject(new TypeError('error due to circular reference'))
6   }
7
8   // 是否已经执行过 onfulfilled 或者 onrejected
9   let consumed = false
10  let thenable
11
12  if (result instanceof LPromise) {
13    if (result.status === 'pending') {
14      result.then(function(data) {
15        resolvePromise(promise2, data, resolve, reject)
16      }, reject)
17    } else {
18      result.then(resolve, reject)
19    }
20    return
21  }
22
23  let isComplexResult = target => (typeof target === 'function' || typeof targ
24
25  // 如果返回的是疑似 Promise 类型
26  if (isComplexResult(result)) {
27    try {
28      thenable = result.then
29      // 如果返回的是 Promise 类型, 具有 then 方法
30      if (typeof thenable === 'function') {
31        thenable.call(result, function(data) {
32          if (consumed) {
33            return
34          }
35          consumed = true
36
37          return resolvePromise(promise2, data, resolve, reject)
38        }, function(error) {
39          if (consumed) {
40            return
41          }
42          consumed = true
43
44          return reject(error)
```

```

45         })
46     }
47     else {
48         resolve(result)
49     }
50
51     } catch(e) {
52         if (consumed) {
53             return
54         }
55         consumed = true
56         return reject(e)
57     }
58 }
59 else {
60     resolve(result)
61 }
62 }
63
64 function LPromise(execute) {
65     this.status = "pending";
66     this.value = null;
67     this.reason = null;
68
69     this.onFulfilledArray = [];
70     this.onRejectedArray = [];
71
72     const resolve = (value) => {
73         queueMicrotask(() => {
74             if(this.status === "pending") {
75                 this.value = value;
76                 this.status = "fulfilled";
77                 this.onFulfilledArray.forEach(func => func(value))
78             }
79         })
80     }
81
82     const reject = (reason) => {
83         queueMicrotask(() => {
84             if(this.status === "pending") {
85                 this.reason = reason;
86                 this.status = "rejected";
87                 this.onRejectedArray.forEach(func => func(reason))
88             }
89         })
90     }
91     // try catch

```

```

92     execute(resolve, reject);
93 }
94
95 LPromise.prototype.then = function(onFulfilled, onRejected) {
96     onFulfilled = typeof onFulfilled === "function" ? onFulfilled: (data) => { r
97     onRejected = typeof onRejected === "function" ? onRejected: (error) => { thr
98
99     let promise2;
100
101     if(this.status === "fulfilled") {
102         return promise2 = new LPromise((resolve, reject) => {
103             queueMicrotask(() => {
104                 try {
105                     // promise1 中 onfulfilled 返回了一个值, 这个值需要被 promise2 ;
106                     let x = onFulfilled(this.value);
107                     resolvePromise(promise2, x, resolve, reject)
108                 } catch(e) {
109                     reject(e)
110                 }
111             })
112         })
113     }
114
115
116     if(this.status === "rejected") {
117         return promise2 = new LPromise((resolve, reject) => {
118             queueMicrotask(() => {
119                 try {
120                     // promise1 中 onfulfilled 返回了一个值, 这个值需要被 promise2 ;
121                     let x = onRejected(this.reason);
122                     resolvePromise(promise2, x, resolve, reject)
123                 } catch(e) {
124                     reject(e)
125                 }
126             })
127         })
128     }
129
130     if(this.status === "pending") {
131         return promise2 = new LPromise((resolve, reject) => {
132             this.onFulfilledArray.push(() => {
133                 try {
134                     let x = onFulfilled(this.value);
135                     resolvePromise(promise2, x, resolve, reject)
136                 } catch(e) {
137                     reject(e)
138                 }

```



```

139         });
140         this.onRejectedArray.push(() => {
141             try {
142                 // promise1 中 onfulfilled 返回了一个值，这个值需要被 promise2 ;
143                 let x = onRejected(this.reason);
144                 resolvePromise(promise2, x, resolve, reject)
145             } catch(e) {
146                 reject(e)
147             }
148         });
149     })
150
151 }
152 }
153
154

```

Promise 的一些理解

链式执行

100 个 promise，10个先执行，每 resolve 一个，加一个进去。形成 stream.

```

1  const promiseArrGenerator = (num) =>
2      new Array(num).fill(0).map((item, index) => () => new Promise((resolve, reje
3          setTimeout(() => {
4              resolve(index)
5          }, Math.random() * 1000)
6      )))
7
8  let arr = promiseArrGenerator(100);
9
10 // arr.map((fn) => {
11 //     fn().then(console.log)
12 // })
13 // Promise.all(arr.map(fn => fn())).then(res => console.log(res))
14
15 // 设计一个 promise Chain 链式调用
16
17 const promiseChain = (arr) => {
18     arr.reduce((proChain, pro) => proChain.then(res => {
19         ~res && console.log(res);
20         return pro()
21     }), Promise.resolve(-1))

```

```
22 }  
23  
24 promiseChain(arr)  
25
```

手动触发

sleep 函数, hark 函数

```
1  const engine = (cb) => {  
2    let _resolve;  
3  
4    new Promise((resolve, reject) => {  
5      _resolve = resolve;  
6    }).then(res => {  
7      cb()  
8    })  
9  
10   return {  
11     start: () => {  
12       _resolve()  
13     }  
14   }  
15 }  
16  
17 let e = engine(() => {  
18   console.log("engine")  
19 })  
20  
21 e.start()
```