es11新特性

# 1.Promise.allSettled

## 这个方法金额Promise.all不一样，Promise.all方法只有数组里面的所有promise都是fulfilled状态，才会返回成功，否则返回失败，执行catch代码，这个方法会获取所有的结果，不管是fulfilled还是rejected，把结果放到一个数组中。也就是说Promise.all方法一定会执行所有的promise对象然后一定走then代码而不走catch它有它特点的应用场景的，如：聚合网站

|  |
| --- |
|  |

## 自己写的案例

|  |
| --- |
| <script>          //Promise.allSettled          //1.封装自己的ajax          function myAjax(url) {              return new Promise(function(resolve,reject) {                  let xhr = new XMLHttpRequest()                  xhr.open("get",url,true)                  xhr.send()                  xhr.onreadystatechange = function () {                      if(xhr.readyState === 4){                          if(xhr.status>=200 && xhr.status<=300){                          resolve(JSON.parse(xhr.responseText))                          } else {                              reject(xhr.responseText)                          }                      }                    }              })            }     let arr = [myAjax("./1.json"),myAjax("./4.json")]  //    Promise.allSettled(arr).then(res=>console.log(res)) //这个方法会执行使用Promise，不管是否执行成功，都会拿到结果     //这个方法返回一个数组，我们来根据数组元素的status属性来过滤      Promise.allSettled(arr).then(res=>{      //    let ret1 = res.filter(item=>item.status=="fulfilled") //过滤出成功的案例      //    console.log(ret1);         let ret2 = res.filter(item=>item.status=="rejected") //过滤出失败的案例         console.log(ret2);      })      </script> |
|  |

# 2.module新增

## 2.1动态导入，这个是重点

|  |
| --- |
|  |

## 课堂小案例模拟用户登录，根据不同用户加载不同的js文件

|  |
| --- |
| <body>      <button id="login">登录</button>      <script type="module">          function adminlogin() {          return "管理员"        }        function userlogin() {           return "用户"        }       function render(role) {          if(role ==="管理员" ){            let res1 = import('./1.js') //这个import函数的返回值是一个Promise          //   console.log(res1);            res1.then((ret)=>console.log(ret.default.name))          } else if(role ==="用户" ){             let res2 = import('./2.js')          //    console.log(res2);             res2.then((ret)=>console.log(ret.default.name))          }       }       //按钮点击事件处理       let btn = document.querySelector('#login')       btn.onclick = () =>{            let role = adminlogin()            console.log(role);            render(role)          //   let role1 = userlogin()          //   console.log(role1);          //   render(role)       }        </script>  </body> |
|  |

## 可以把代码优化一下，改为async await的方式

|  |
| --- |
| <body>      <button id="login">登录</button>      <script type="module">          function adminlogin() {          return "管理员"        }        function userlogin() {           return "用户"        }      async function render(role) {          if(role ==="管理员" ){            let res1 = await import('./1.js') //这个import函数的返回值是一个Promise            console.log(res1.default.name);          } else if(role ==="用户" ){             let res2 =await import('./2.js')             console.log(res2.default.name);            }       }       //按钮点击事件处理       let btn = document.querySelector('#login')       btn.onclick = () =>{          //   let role = adminlogin()          //   console.log(role);          //   render(role)            let role1 = userlogin()            console.log(role1);            render(role1)       }        </script>  </body> |
|  |

### 这个功能以后非常常用

## 2.2 import.meta

|  |
| --- |
|  |

## 小案例，在2.js里面添加import.meta的输出

|  |
| --- |
| console.log("2.js loaded....",import.meta);  export default {      name:"用户模块"  } |

### dynamic.html

|  |
| --- |
| <body>      <button id="login">登录</button>      <script type="module">          function adminlogin() {          return "管理员"        }        function userlogin() {           return "用户"        }      async function render(role) {          if(role ==="管理员" ){            let res1 = await import('./1.js') //这个import函数的返回值是一个Promise,最好配合async await来使用            console.log(res1.default.name);          } else if(role ==="用户" ){             let res2 =await import('./2.js')             console.log(res2.default.name);            }       }       //按钮点击事件处理       let btn = document.querySelector('#login')       btn.onclick = () =>{          //   let role = adminlogin()          //   console.log(role);          //   render(role)            let role1 = userlogin()            console.log(role1);            render(role1)       }        </script>  </body> |
|  |

## 2.3 export \* as obj from xx.js,在一个模块里面导出另外一个模块里面的全部东西

|  |
| --- |
|  |

## 小案例

### 我们新建一个3.js，内容如下

|  |
| --- |
| console.log("inside 3.js");  export \* as js1 from './1.js' |

### 1.js的内容

|  |
| --- |
| console.log("1.js loaded....");  export function myfun(){      console.log("myfun of 1.js");  }  export default {      name:"管理员模块"  } |

### 我们新建一个dynamic2.html,内容如下

|  |
| --- |
| <!DOCTYPE html>  <html lang="en">  <head>      <meta charset="UTF-8">      <meta name="viewport" content="width=device-width, initial-scale=1.0">      <title>Document</title>  </head>  <body>      <script type="module">          import \*as js1 from './3.js'          console.log(js1);      </script>  </body>  </html> |
|  |

#### 也就是说，一个模块可以把另外一个模块导出的东西全部导入再打包导出为一个对象，然后第三个文件导入这个文件就导入了第一个模块的使用导出的功能，这种方式的应用场景就是你需要使用某一个模块，但是你不能去修改人家的内容，你可以在你的模块中把这个模块的所有导出的内容导出，而且你自己的模块也有内容导出的场景

|  |
| --- |
|  |

### 也可以用解构表达式来导入

|  |
| --- |
|  |

# 3.String.matchAll()

## 课堂案例1

|  |
| --- |
|  |
| <script>          let str = `          <ul>              <li>111</li>              <li>222</li>              <li>333</li>              <li>444</li>          </ul>`          //以前的做法          let reg = /<li>(.\*)<\/li>/g          console.log(str.match(reg));     //['<li>111</li>', '<li>222</li>', '<li>333</li>', '<li>444</li>']          //str.matchAll方法          let it = str.matchAll(reg)          // console.log(it);          //使用for..of遍历          for (let i of it){              console.log(i);          }      </script>  </body> |
|  |

## 其实这个迭代器可以用...展开变为数组

|  |
| --- |
| let str = `          <ul>              <li>111</li>              <li>222</li>              <li>333</li>              <li>444</li>          </ul>`          //以前的做法          let reg = /<li>(.\*)<\/li>/g          console.log("match的结果:");          console.log(str.match(reg));     //['<li>111</li>', '<li>222</li>', '<li>333</li>', '<li>444</li>']          console.log("matchAll方法的结果:");          //str.matchAll方法          let it = str.matchAll(reg)          // console.log(it);          //使用for..of遍历          // for (let i of it){          //     console.log(i);          // }          console.log([...it]); //可迭代对象展开变为数组 |

## 可见matchAll方法能够拿到更多东西，不仅仅拿到li标签还可以把li标签里面的文本单独拿出来

### 注意：用正则的exec也能够捕获到我们需要的内容，不过比较麻烦需要调用4次

|  |
| --- |
|  |
|  |

# 4.BigInt

|  |
| --- |
|  |

## 注意：有一个第三方模块叫做json-bigint可以处理BigInt： [sidorares/json-bigint: JSON.parse/stringify with bigints support (github.com)](https://github.com/sidorares/json-bigint)， 这个库只能在nodejs或者vue，react的环境中

|  |
| --- |
|  |

### 上面是用es6的模块方式使用，需要在package.json里面添加type:”module”

### 上面的库以前只能将很大的数字转化为字符串，现在他也能够把它转化为BigInt

|  |
| --- |
|  |
|  |
|  |
|  |

## Bigint参考文档

## 一、bigint是什么

bigint类型是ES11提供的一个新的数据类型，由于number类型只能在[Number.MIN\_SAFE\_INTEGER, Number.MAX\_SAFE\_INTEGER]范围内是安全的，超出了这个范围，精度会丢失，所以bigint出现了，表示任意大的整数。

## 二、如何定义bigint

### 1、在数字后加上n

const num = 1n

console.log(typeof num) // bigint

console.log(Object.prototype.toString.call(num)) // [object BigInt]

### 2、使用Bigint函数

Bigint函数中传的必须是整数值或者字符串类型的整数值

const num = BigInt('1')

console.log(typeof num) // bigint

console.log(Object.prototype.toString.call(num)) // [object BigInt]

## 三、bigint和number的区别

### 1、不能使用Math中的方法，Math仅用于number类型，不支持bigint

Math.ceil(1.2) // 2

Math.ceil(BigInt(1)) // 报错

### 2、不能和number混合运算，必须要转换成同一种类型

const a = 1

const b = 2n

const c = BigInt(a) + b // 必须要转为bigint或者number

const d = a + Number(b)

const e = a + b // 报错

console.log(c) // 3n

console.log(d) // 3

### 3、bigint在转换为number时可能会精度丢失

不建议将bigint转换为number，并且仅在超过number最大安全范围时才使用bigint

### 4、number可以是小数，bigint只能是整数

console.log(1.2n) // 报错

### 5、bigint不支持单目运算符+

+号可以当做Number()使用：

console.log(+'1') // 1

console.log(+[]) // 0

console.log(+true) // 1

console.log(+3) // 3

console.log(+null) // 0

console.log(+undefined) // NaN

但是+号不能将bigint转换为number，必须要使用Number()：

console.log(+BigInt(1)) // 报错

### 6、bigint和number宽松相等，严格不等

const a = 1n

const b = 1

console.log(a == b) // true

console.log(a === b) // false

## 四、bigint和number相同点

### 1、bigint可以进行+、-、\*、/、%、\*\*运算

但是要注意bigint的除法运算中，小数点后面的值会被抹去，向下取整

const a = 1n

const b = 2n

console.log(a + b) // 3n

console.log(a - b) // -1n

console.log(a \* b) // 2n

console.log(a / b) // 0n, not 0.5n

console.log(a % b) // 1n

console.log(b \*\* b) // 4n

### 2、bigint可以相互之间比较，和number比较规则一致，也可以用来和number比较

console.log(1n < 2n) // true

console.log(1n < 2) // true

console.log(1n <= 1) // true

### 3、bigint和number可以混在一个数组中并排序

const mixed = [4n, 6, -12n, 10, -12, 6n, -13, -13n, 4, 0, 0n]

console.log(mixed.sort()) // [-12n, -12, -13, -13n, 0, 0n, 10, 4n, 4, 6, 6n]

### 4、bigint转换为boolean时和number表现一样

console.log(Boolean(1n)) // true

console.log(Boolean(0n)) // false

console.log(!!1n) // true

console.log(!!0n) // false

console.log(!1n) // false

console.log(!0n) // true

console.log(0n && 1n) // 0n

console.log(0n || 1n) // 1n

if (1n) {

console.log('真') // 打印 真

} else {

console.log('假')

}

### 5、bigint转换为string时和number表现一样

console.log(1 + '') // 1

console.log(1n + '') // 1

console.log(String(1n)) // 1

## 五、bigint使用场景

number类型在超过最大安全数时精度丢失，此时bigint的作用就体现出来了

const a = Number.MAX\_SAFE\_INTEGER

console.log(a + 1 === a + 2) // true

const b = BigInt(a)

console.log(b + 1n === b + 2n) // false

# 5.顶层对象globalThis

|  |
| --- |
|  |

## 在浏览器环境在globalTHis等价于Window顶层对象或者self对象，在nodejs中globalThis等价于global顶层对象

## globalTHis参考文档

一、globalThis对于WEB(浏览器)环境下，js中的全局变量是window/self，相信大家对于window了解的比较多，也比较熟悉，小编也是今天才知道self的这个对象，从目前来看，可以认为是一样的。下面是延时实例

setTimeout(() => {

console.log('lilei')

},1000) // lilei

因为上面的例子是在全局下的方法，如果写全的话，应该是这样

window.setTimeout(() => {

console.log('lilei')

},1000) // lilei

self.setTimeout(() => {

console.log('lilei')

},1000) // lilei

同样，我们可以自定义一个获取不同环境下的全局变量方法，就像这样

const getGlobal = () => {

if(typeof self !== 'undefined'){

return self

}

if(typeof window !== 'undefined'){

return window

}

if(typeof global !== 'undefined'){

return global

}

throw new Error('无法找到全局对象')}

const testGlobal = getGlobal()console.log(testGlobal)

在es11引入globalThis后，上面的一堆代码就可以简写成这样

console.log(globalThis)

二、可选链 之前小编在看别人代码的时候发现过这个符号，但是在网上搜索的时候，根本不知道怎么去搜索，也就临时搁下了，直到看到了可选链的概念，我才恍然大悟。先举一个实际的例子，我们在调用对象的属性或者方法的时候，很多时候我们不知道后端传递过来的数据是不是确切有指定属性，这时候，我们的代码就会写成这样：

const user = {

address:{

street:'No.1 Street',

getNum(){

return '88号'

}

}}

const street = user && user.address && user.address.streetconsole.log(street) // No.1 Street

const streetNum= user && user.address && user.address.getNum && user.address.getNum() // 是否存在之后才去调用console.log(streetNum) // 88号

# 6.空合并运算符

|  |
| --- |
|  |

## 注意，只有左边的值是null或者undefined，才会返回右边的值，其他值就捕获返回右边的值

## 小案例

|  |
| --- |
| <body>      <script>          //??          console.log(null??"默认值");          console.log(undefined??"默认值");          console.log(0??"默认值");          console.log(''??"默认值");          console.log(false??"默认值");      </script>  </body>  </html> |
|  |

# 7.可选链

|  |
| --- |
|  |

## 课堂案例，比较2种情况

|  |
| --- |
| <script>          //?.只要前面的为null或者undefined就捕获执行后面的          let obj = {              name:"Jackline",              intruduction:0            }          console.log("obj...");          console.log(obj && obj.location && obj.location.city); //undefined          console.log(obj?.location?.city); //undefined          console.log("obj2...");          let obj2 = {              name:"Jackline",              intruduction:0,              location:{                  city:"dalian"              }          }          console.log(obj2 && obj2.location && obj2.location.city); //dalian          console.log(obj2?.location?.city); //dalian      </script> |
|  |

### 可选链可以和??一起使用，表示当所有的可选链都为null或者undefined就返回我们的默认值

|  |
| --- |
| //?.只要前面的为null或者undefined就捕获执行后面的          let obj = {              name:"Jackline",              intruduction:0            }          console.log("obj...");          console.log(obj && obj.location && obj.location.city); //undefined          console.log(obj?.location?.city); //undefined          console.log(obj?.location?.city??"这里没有loaction属性"); // |
|  |

## 空合并操作符和可选链参考文档

## 空值合并运算符

**空值合并操作符（??）** 是一个逻辑操作符，当左侧的操作数为 null 或者 undefined 时，返回其右侧操作数，否则返回左侧操作数。

与逻辑或操作符（||）不同，逻辑或操作符会在左侧操作数为假值时返回右侧操作数。也就是说，如果使用 || 来为某些变量设置默认值，可能会遇到意料之外的行为。比如为假值（例如，'' 或 0）时。

const x = null ?? 'default string';

console.log(x);

// expected output: "default string"

const y = 0 ?? 15;

console.log(y);

// expected output: 0

## 使用空值合并操作符

在这个例子中，我们使用空值合并操作符为常量提供默认值，保证常量不为 null 或者 undefined。

const nullValue = null;

const emptyText = ""; // 空字符串，是一个假值，Boolean("") === false

const someNumber = 15;

const x = nullValue ?? "x 的默认值";

const y = emptyText ?? "y 的默认值";

const z = someNumber ?? 0;

console.log(x); // "x 的默认值"

console.log(y); // ""（空字符串虽然是假值，但不是 null 或者 undefined）

console.log(z); // 15

## 为变量赋默认值

以前，如果想为一个变量赋默认值，通常的做法是使用逻辑或操作符（||）：

let x;

// foo is never assigned any value so it is still undefined

let someDummyText = x || 'Hi!';

然而，由于 || 是一个布尔逻辑运算符，左侧的操作数会被强制转换成布尔值用于求值。任何假值（0， ''， NaN， null， undefined）都不会被返回。这导致如果你使用0，''或NaN作为有效值，就会出现不可预料的后果。

let count = 0;

let a = "";

let x = count || 15;

let y = a || "Hello!";

console.log(x); // 15，而不是 0

console.log(y); // "Hello!"，而不是 ""

空值合并操作符可以避免这种陷阱，其只在第一个操作数为 null 或 undefined 时（而不是其它假值）返回第二个操作数：

let a = ''; // An empty string (which is also a falsy value)

let x = a || 'Hello world';

console.log(x); // Hello world

let y = a ?? 'Hi!';

console.log(y); // '' (as myText is neither undefined nor null)

## 短路

与 OR 和 AND 逻辑操作符相似，当左表达式不为 null 或 undefined 时，不会对右表达式进行求值。

function A() { console.log('函数 A 被调用了'); return undefined; }

function B() { console.log('函数 B 被调用了'); return false; }

function C() { console.log('函数 C 被调用了'); return "foo"; }

console.log( A() ?? C() );

// 依次打印 "函数 A 被调用了"、"函数 C 被调用了"、"foo"

// A() 返回了 undefined，所以操作符两边的表达式都被执行了

console.log( B() ?? C() );

// 依次打印 "函数 B 被调用了"、"false"

// B() 返回了 false（既不是 null 也不是 undefined）

// 所以右侧表达式没有被执行

## 不能与 AND 或 OR 操作符共用

将 ?? 直接与 AND（&&）和 OR（||）操作符组合使用是不可取的。应当是因为空值合并操作符和其他逻辑操作符之间的运算优先级/运算顺序是未定义的,这种情况下会抛出 SyntaxError 。

null || undefined ?? "foo"; // 抛出 SyntaxError

true || undefined ?? "foo"; // 抛出 SyntaxError

但是，如果使用括号来显式表明运算优先级，是没有问题的：

(null || undefined ) ?? "foo"; // 返回 "foo"

## 与可选链式操作符（?.）的关系

空值合并操作符针对 undefined 与 null 这两个值，可选链式操作符（?.） 也是如此。在这访问属性可能为 undefined 与 null 的对象时，可选链式操作符非常有用。

let foo = { someFooProp: "hi" };

console.log(foo.someFooProp?.toUpperCase()); // "HI"

console.log(foo.someBarProp?.toUpperCase()); // undefined

## 判断为' '/null/undefined的用法

if((value??'')!==''){

}

// 取代

if(value !== null && value !== undefined && value !== ''){

}