# 以前课笔记程链接

**1-12节：**[**es6课堂笔记1-12节**](es6学习笔记1-12节.docx)

# 11.es6迭代器

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

### 注意：实现了迭代器的对象可以用...扩展变为数组

|  |
| --- |
| <body>      <ul>          <li>王祖贤</li>          <li>张曼玉</li>          <li>佘诗曼</li>      </ul>   <script>      let lis= document.querySelectorAll("li")      console.log(lis,[...lis]);// [...lis]就是一个数组     </script>  </body> |
|  |

## 注意：这么写是不行的

|  |
| --- |
|  |
|  |

### 也就是说，对象默认不支持for..of,但是对象默认支持for..in,获取到的是对象的键

|  |
| --- |
| //对象默认不支持for..of遍历但是对象默认支持for in遍历，获取到的key      let nameObj3 = {          name:"Jackline",          gender:'Female',          age:18,        }      for (let i in nameObj3) {          console.log(`${i}:${nameObj3[i]}`);      } |
|  |

## 1）for..of和for...in的区别

|  |
| --- |
| for..of获取的是数组元素的值，for...in获取的是元素的索引 |

## 2)遍历的原理

### 数组之所以能够用上面的方法来遍历，是因为它有一个Symbol(iterator)接口

|  |
| --- |
|  |

注意：Symbol有一个属性很特殊，就是Symbol.iterator,如果一个对象有这个属性而且他是方法，可以这么使用obj[Symbol.iterator]()

|  |
| --- |
| let obj = {          name:"Jade",          age:18,          [Symbol.iterator]:function(){              console.log(`hello my name is ${this.name},i am ${this.age} years old`);          }      }      obj[Symbol.iterator]() |
|  |

迭代器就是基于这个，

### obj[Symbol.iterator]()返回的是一个迭代器，它有一个next()方法可以获取元素

|  |
| --- |
| <script>      let arr = ["Java","cpp","Python"]  console.log(arr[Symbol.iterator]());  //获取迭代器      let iter = arr[Symbol.iterator]()      console.log(iter.next().value);//获取元素      console.log(iter.next().value);      console.log(iter.next().value);        </script> |

# 12.es的Set

## 1）什么是Set

[ES6](https://so.csdn.net/so/search?q=ES6&spm=1001.2101.3001.7020)**提供了新的数据结构Set。它类似于数组，但是成员的值都是唯一的，没有重复的值。Set本身是一个构造函数，用来生成Set数据结构，set是无序的。**

## 2）Set的特点

#### 1、类似于数组，但是成员的值都是唯一的，没有重复的值

#### 2. set是无序的，添加元素的时候不会自动排序

#### 3.set的元素不能用下标获取

## 3）Set的基本使用

### 1>创建Set

Set 函数可以接受一个数组（或者具有 iterable 接口的其他数据结构）作为参数，用来初始化

|  |
| --- |
| let s = new Set();  [2, 3, 5, 4, 5, 2, 2].map(x => s.add(x));  for (let i of s) {  console.log(i); //2，3，5，4  } |
| **let** set = **new** **Set**([1, 2, 3, 3, 4]);  console.**log**(set); *// Set(4) {1, 2, 3, 4}* |
| set = **new** **Set**(document.**querySelectorAll**('div')) |

#### 注意，不能直接给Set构造函数传递几个数字来创建set需要传递一个可迭代对象，下面的代码会报错

|  |
| --- |
| let set2 = new Set(3,4,5,6) //错误：不能直接给Set构造函数传递几个数字来创建set需要传递一个可迭代对象   console.log(set2); |
|  |

#### 函数的参数列表可以转换为set

|  |
| --- |
| // 函数的参数列表也可以转换为set          function testSet() {              let set = new Set(arguments)              console.log(set);//Set(3) {10, 20, 30}          }          testSet(10,20,30) |

### 2>元素的操作方法

 add(value)：添加某个值，返回 Set 结构本身。所以可以链式调用

. delete(value)：删除某个值，返回一个布尔值，表示删除是否成功。

 has(value)：返回一个布尔值，表示该值是否为 Set 的成员。

 clear()：清除所有成员，没有返回值。

|  |
| --- |
| let set = new Set(); console.log(set.add(1).add(2)); // Set [ 1, 2 ] console.log(set.delete(2)); // true console.log(set.has(2)); // false console.log(set.clear()); // undefined console.log(set.has(1)); // false |

### 3>遍历Set

** 直接for...of**

**Set.prototype.keys()：返回键名的遍历器**

** Set.prototype.values()：返回键值的遍历器**

** Set.prototype.entries()：返回键值对的遍历器**

** Set.prototype.forEach()：使用回调函数遍历每个成员**

|  |
| --- |
| //set的遍历          let mySet =new Set([2,3,4,5,6])            // 遍历1，for..of直接遍历          for (const i of mySet) {              console.log(i); //set对象可以直接用for...of遍历          } |
|  |
| let mySet =new Set([2,3,4,5,6])            // 遍历2，for..of遍历keys()          for (const i of mySet.keys()) {              console.log(i); //set对象可以用for...of遍历          } |
|  |
| let mySet =new Set([2,3,4,5,6])          // 遍历3，for..of遍历values()      for (const i of mySet.values()) {              console.log(i);      } |
|  |
| let mySet =new Set([2,3,4,5,6])           //遍历4.for..of遍历entries(),每一个entry其实是一个数组第一个元素是key，第二个元素是value   for(let e of mySet.entries()){      console.log(e[0],e[1]);   } |
|  |
| let mySet =new Set([20,3,4,5,6])          // console.log(mySet.keys());          // for (let i in mySet) { //set不能使用for..in循环，因为他没有索引          //     console.log(i);          // }          // // 遍历1，for..of直接遍历          // for (const i of mySet) {          //     console.log(i); //set对象可以直接用for...of遍历          // }          // // 遍历2，for..of遍历keys()          // for (const i of mySet.keys()) {          //     console.log(i); //set对象可以用for...of遍历          // }          // // 遍历3，for..of遍历values()          // for (const i of mySet.values()) {          //     console.log(i);          // }         //遍历4.for..of遍历entries(),每一个entry其实是一个数组第一个元素是key，第二个元素是value      //    for(let e of mySet.entries()){      //      console.log(e[0],e[1]);      //    }         //遍历5 forEach方法         mySet.forEach(item=>console.log(item)) |
|  |

注意：js的Set和Java，kotlin的Set有点不一样，它有key，key和value是一样的，而Java等等的Set是没有key的

### 4>Set和数组的转换

#### set-》数组：arr = [...set] 或者 arr = Array.from(set)

#### 数组-》set： set = Set(arr)

### Set的主要用处就是去除重复元素，然后转化为数组，这样子比较好用

### 面试题：复杂数组去重

### 有这么一个数组，我们需要把它去除重复元素

|  |
| --- |
|  |

### 我们来写一个方法，需要将对象转化为json字符串和结合set对象来去重

|  |
| --- |
|  |

### 效果是ok的

|  |
| --- |
|  |

### 即使有undefined和NaN也没有问题

|  |
| --- |
|  |
|  |

### 注意：即使你用二维数组来构造set对象，他的可以和value都是一样的

|  |
| --- |
| //从二维数组创建set，会有很奇怪的key          let set1 = new Set([["name","Mark"],["gender","Male"],["age",18]])          console.log(set1);          set1.forEach((k,v)=>console.log(`${k}:${v}`)) |
|  |

# 13.es的Map

## 1.Map是什么

Map数据结构类似于对象，也是键值对的集合，但是键的范围不限于字符串，各种类型的值（包括对象）都可以当做键。

Map结构提供了“值-值”的对应，是更完善的Hash结构实现。Map可以作为构造函数。

|  |
| --- |
|  |

### map对象也是可以用...扩展为一个二维数组的

|  |
| --- |
|  |
|  |

### 注意：如果想用对象作为map的key，需要先用一个变量吧对象接收了，然后使用这个变量作为map的key，否则不能获取这个对象键的值

|  |
| --- |
|  |
|  |

## 2.Map的创建

### 1》创建一个空的Map对象，然后使用set方法赋值

|  |
| --- |
| let map = new Map()          map.set("name","Kenny")          map.set("position","Boss")          map.set("money",99999999999999.99)          console.log(map); //Map(3) {'name' => 'Kenny', 'position' => 'Boss', 'money' => 99999999999999.98} |
|  |

### 2》直接使用二维数组构造

|  |
| --- |
| //方法2，用二维数组来构造map          let map1 = new Map([["name","Jade"],["gender","female"],["age",20]])          console.log(map1); //Map(3) {'name' => 'Jade', 'gender' => 'female', 'age' => 20} |
|  |

### 3》方法3，使用set对象创建

|  |
| --- |
| //方法3，使用set创建          let map2 = new Map(new Set([["code",100],["result",'ok'],['method','get']]))          console.log(map2); |
|  |

## 3.Map操作元素

### 1》设置值，用set(key,value)

|  |
| --- |
| let map2 = new Map(new Set([["code",100],["result",'ok'],['method','get']]))         //console.log(map2);          //2.元素操作          //1)设置值，用set          map2.set("site","google.com")          console.log(map2); |
|  |

### 2>获取值，用get(key)

|  |
| --- |
| let map2 = new Map(new Set([["code",100],["result",'ok'],['method','get']]))          // console.log(map2);          //2.元素操作          //1)设置值，用set          map2.set("site","google.com")          // console.log(map2);          //2.获取值，用get          console.log(map2.get('site')); //google.com |
|  |

### 3>删除元素，使用delete

|  |
| --- |
| let map2 = new Map(new Set([["code",100],["result",'ok'],['method','get']]))          // console.log(map2);          //2.元素操作          //1)设置值，用set          map2.set("site","google.com")          // console.log(map2);          //2.获取值，用get          // console.log(map2.get('site')); //google.com          //3.删除元素，使用delete方法          map2.delete("site")          console.log(map2);//Map(3) {'code' => 100, 'result' => 'ok', 'method' => 'get'} |
|  |

### 4>判断是否包含某个key，用has方法

|  |
| --- |
| let map2 = new Map(new Set([["code",100],["result",'ok'],['method','get']]))          // console.log(map2);          //2.元素操作          //1)设置值，用set          map2.set("site","google.com")          // console.log(map2);          //2.获取值，用get          // console.log(map2.get('site')); //google.com          //3.删除元素，使用delete方法          map2.delete("site")          // console.log(map2);//Map(3) {'code' => 100, 'result' => 'ok', 'method' => 'get'}          //4.判断是否包含某个key，用has方法          console.log(map2.has('site')); //false,因为这个key已经被删除 |
|  |

### 5》清空map，使用clear方法

|  |
| --- |
| let map2 = new Map(new Set([["code",100],["result",'ok'],['method','get']]))          // console.log(map2);          //2.元素操作          //1)设置值，用set          map2.set("site","google.com")          // console.log(map2);          //2.获取值，用get          // console.log(map2.get('site')); //google.com          //3.删除元素，使用delete方法          map2.delete("site")          // console.log(map2);//Map(3) {'code' => 100, 'result' => 'ok', 'method' => 'get'}          //4.判断是否包含某个key，用has方法          // console.log(map2.has('site')); //false,因为这个key已经被删除          //5.清空map，使用clear方法          map2.clear()          console.log(map2); //Map(0) {size: 0} |
|  |

## 4.Map的遍历

### 1）直接for..of遍历，返回一个个的数组，数组第一个值是key，第二个值是value

|  |
| --- |
| //3.map的遍历          let map3 =  new Map(new Set([["code",100],["result",'ok'],['method','get'],['site','facebook.com']]))          //1.for..of直接遍历，返回一个个的数组，数组第一个值是key，第二个值是value          for (const i of map3) {              console.log(i);          } |
|  |

### 2.遍历keys，得到所有的键

|  |
| --- |
| let map3 =  new Map(new Set([["code",100],["result",'ok'],['method','get'],['site','facebook.com']]))          // console.log(map3['site']);          //1.for..of直接遍历，返回一个个的数组，数组第一个值是key，第二个值是value          // for (const i of map3) {          //     console.log(i);          // }          //2.遍历keys          for(let k of map3.keys()){              console.log(k); //返回所有的key          } |
|  |

### 3.遍历values得到所有的值

|  |
| --- |
| let map3 =  new Map(new Set([["code",100],["result",'ok'],['method','get'],['site','facebook.com']]))            //3.values得到所有的值          for(let v of map3.values()){              console.log(v);          } |
|  |

### 4.entries方法，效果跟直接for..of是一样的

|  |
| --- |
| //3.map的遍历          let map3 =  new Map(new Set([["code",100],["result",'ok'],['method','get'],['site','facebook.com']]))          // console.log(map3['site']);          //1.for..of直接遍历，返回一个个的数组，数组第一个值是key，第二个值是value          // for (const i of map3) {          //     console.log(i);          // }          //2.遍历keys          for(let k of map3.keys()){              // console.log(k); //返回所有的key          }          //3.values得到所有的值          for(let v of map3.values()){              // console.log(v);          }          //4.entries()方法          for(let e of map3.entries()){              console.log(e);          } |
|  |

### 5.forEach方法，获取到的第一个是key，第二个才是key，需要注意

|  |
| --- |
| //3.map的遍历          let map3 =  new Map(new Set([["code",100],["result",'ok'],['method','get'],['site','facebook.com']]))          // console.log(map3['site']);          //1.for..of直接遍历，返回一个个的数组，数组第一个值是key，第二个值是value          // for (const i of map3) {          //     console.log(i);          // }          //2.遍历keys          for(let k of map3.keys()){              // console.log(k); //返回所有的key          }          //3.values得到所有的值          for(let v of map3.values()){              // console.log(v);          }          //4.entries()方法          for(let e of map3.entries()){              // console.log(e);          }          //5.forEach方法，注意获取到的第一个是value，第二个才是key          map3.forEach((value,key)=>console.log(value,key)) |
|  |

### 注意：

### 1.在js中Set，Map和数组都是有entries()方法的，前两个的每一个entry是一个有key和value组成的数组，数组的entry是索引和元素值组成的数组

### 2.js中的map不能使用map[key]的方法来获取值，获取不到，只能使用get方法这个和我Java是不一样的

|  |
| --- |
| let map3 =  new Map(new Set([["code",100],["result",'ok'],['method','get'],['site','facebook.com']]))          console.log(map3['site']); |
|  |

# 14.es6中的Proxy

## 课堂笔记

|  |
| --- |
|  |

### Proxy的原理

### 在Proxy出来之前，我们可以使用Object.defineProperty()方法来给对象添加属性，这个比直接给对象添加属性要好，因为你可以在里面定义属性的get和set方法

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

### 这就是一些响应式框架的运行原理，不过这个方法每一次只能拦截一个属性，效率不高，而且他只能拦截对象，不能拦截数组

### 现在我们来学习Proxy

### 1.proxy的基本用法

|  |
| --- |
| <body>      <div id="box"></div>      <script>          //2.Proxy基本使用        let obj={}        let proxy = new Proxy(obj,{          get(target,key){              console.log(`getting value of ${key} `);              return target[key]          },          set(target,key,value){              console.log(`setting ${value} to ${key}`);              target[key] = value          }        })      </script>  </body> |
|  |

#### 注意：proxy对象还可以有一个has方法，用来判断是否有这个属性，其实没有什么作用

### 2.proxy拦截Set和Map，注意和拦截对象的代码是不一样的，如果使用下面的代码。会出现一个错误

|  |
| --- |
| <body>      <div id="box"></div>      <script>          let set = new Set()        let proxy = new Proxy(set,{          get(target,key){              console.log(`getting value of ${key} `);              return target[key]          },          set(target,key,value){              console.log(`setting ${value} to ${key}`);              //修改dom              if(key === 'data'){                box.innerHTML = value              }              target[key] = value          }        })       //proxy拦截Set和Map        </script>  </body> |
|  |

#### set是有方法的，proxy默认只是处理属性，方法需要我们自己来处理，

|  |
| --- |
| <body>      <div id="box"></div>      <script>        //proxy拦截Set        let set = new Set()        let proxy = new Proxy(set,{          get(target,key){              console.log(`getting value of ${key} `);              //判断如果是方法，修正this指向              let value = target[key] //先把value取出来              if (value instanceof Function) {                 return value.bind(target) //将this指向target              }              return value //如果是属性就直接返回          },          set(target,key,value){              console.log(`setting ${value} to ${key}`);              //修改dom              if(key === 'data'){                box.innerHTML = value              }              target[key] = value          }        })          </script>  </body> |
|  |

#### 拦截Map也是一样的道理

|  |
| --- |
| //Proxy拦截Map       let map = new Map() //map不能使用对象来构建，因为对象默认是包括迭代的       let mapProxy = new Proxy(map,{          get(target,key){            console.log(`获取${key}的值`);            let value = target[key] //先把value取出来              if (value instanceof Function) {                 return value.bind(target) //将this指向target              }               return value          },          set(target,key,value){              console.log(`把值${value}设置进键${key}里面`);              target[key] = value          }       })         mapProxy.set("name","Boss")       mapProxy.get("name") |
|  |

### 注意，拦截set和map的时候，无论你做什么操作都只会重复get方法，这个和对象是不一样的，对应set方法这些数据结构有自己默认的方法，你对set方法的修改不起作用，如果你硬是要改，需要使用Reflact对象来实现

## Proxy参考文档1

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **ES新特性之Proxy对象**    在ES2015的标准中新增了一个Proxy，用于修改某些操作的默认行为，等同于在语言层面作出的修改，所以说这是属于一种**元编程** ，（meta programming），即对编程语言进行编程。  Proxy对象用于创建一个对象的“代理”，从而实现基本能操作的拦截和自定义(如属性查找、赋值、[枚举](https://so.csdn.net/so/search?q=%E6%9E%9A%E4%B8%BE&spm=1001.2101.3001.7020)、函数调用等)。  通俗一点的解释就是，房子比作原始对象，中介比作Proxy，我们租房可以直接找原始房源，也可以通过中介，中介在其中可以屏蔽原始信息。  Proxy的语法结构如下所示：  **const p = new Proxy(target, handler)**  参数说明：   * target：要使用Proxy 代理的目标对象(可以是任何类型的对象，包括原生数组，函数，甚至另一个代理)。 * handler：这个参数是一个配置对象，对于每一个被代理的操作，都需要提供一个对应的处理函数。   **Proxy可拦截的操作**  Proxy可拦截的操作如下表所示：   | **拦截 ⽅法** | **触发⽅式** | | --- | --- | | get(target, propKey, receiver) | 读取某个属性 | | set(target, propKey, value, receiver) | 写⼊某个属性 | | has(target, propKey) | in操作符 | | deleteProperty(target, propKey) | delete操作符 | | getPrototypeOf(target) | Object.getPropertypeOf() | | setPrototypeOf(target, proto) | Object.setPrototypeOf() | | isExtensible(target) | Object.isExtensible() | | preventExtensions(target) | Object.preventExtensions() | | getOwnPropertyDescriptor(target, propKey) | Object.getOwnPropertyDescriptor() | | defineProperty(target, propKey, propDesc) | Object.defineProperty() | | ownKeys(target) | Object.keys() 、Object.getOwnPropertyNames()、Object.getOwnPropertySymbols() | | apply(target, thisArg, args) | 调⽤⼀个函数 | | construct(target, args) | ⽤ new 调⽤⼀个函数 |   以上就是Proxy可拦截的所有操作。  **Proxy实例的方法**  下面我们就对上面那些拦截操作进行逐一演示  **get()**  get方法用于拦截某个属性的读取操作，该方法可以接受是三个参数，分别是目标对象、被获取的属性名和Proxy实例本身(可选)。  下面是对get的简单用法，如果我们访问一个对象中未定义的属性，可以通过get使其抛出一个异常。  const person = {  name: '一碗周'  }  const proxy = new Proxy(person, {  /\*\*  \* @param {object} target 目标对象  \* @param {string} propKey 被获取的属性名  \* @param {object} receiver Proxy或者继承Proxy的对象  \*/  get (target, propKey, receiver) {  if (propKey in target) {  return target[propKey]  } else {  throw new ReferenceError(`属性：${propKey} 不存在`)  }  }  })  proxy.name // 一碗周  proxy.hobby // 抛出异常  get方法是可以继承的，看下面这段代码  const person = {  name: '一碗周'  }  const proxy = new Proxy(person, {  get (target, properKey, receiver) {  console.log(`获取${properKey}属性的值：${target[properKey]}`)  return target[properKey]  }  })  // 通过proxy创建一个新的对象  const obj = Object.create(proxy)  obj.name // 获取name属性的值：一碗周  上面代码中，拦截操作定义在Prototype对象上面，所以如果读取obj对象继承的属性时，拦截会生效。  我们还可以通过get方法去实现数据读取负数的索引，实现代码如下：  function createArray (...elements) {  let handler = {  get (target, properKey, receiver) {  // 1. 缓存传入的索引值  let index = Number(properKey)  if (index < 0) {  // 例如 target 的长度为7，传递的 index 为 -1, 最后的 properKey 等于 6 即最后一项  properKey = String(target.length + index)  }  /\*\*  \* Reflect.get() 的作用是读取一个对象中的值  \*/  return Reflect.get(target, properKey, receiver)  }  }  let target = []  target.push(...elements)  return new Proxy(target, handler)  }  const arr = createArray(1, 2, 3, 4, 5)  console.log(arr[-1]) // 5  值得注意的是，如果一个属性是不可配置（configurable）且不可写（writable），则Proxy不能修改该属性，否则就会抛出异常。  **set()**  set方法用于拦截某个对象的修改操作，该方法接受四个参数分别是目标对象、被获取的属性名、修改后的值和Proxy实例本身(可选)。  下面这个例子就规定了修改某个属性必须是某个类型的值，否则就会抛出异常  const person = {  name: '一碗周'  }  // 1. 根据指定条件修改name属性  const proxy = new Proxy(person, {  /\*\*  \* @param {object} target 目标对象  \* @param {string} propKey 被获取的属性名  \* @param {any} value 修改后的值  \* @param {object} receiver Proxy或者继承Proxy的对象  \* @returns {boolean} 返回true表示成功，严格模式下返回false则会抛出一个异常  \*/  set (target, propKey, value, receiver) {  // 规定修改name属性只能修改为一个字符串，否则就会抛出异常  if (propKey === 'name' && typeof value !== "string") {  throw new TypeError('修改的name并不是一个string')  }  target[propKey] = value  return true  }  })  // proxy.name = 1 // 抛出异常  proxy.name = '一碗周' // 修改成功  我们该可以利用set方法进行数据绑定，如果对象发生改变，就自动更新我们的DOM。  示例代码如下：  <!DOCTYPE html>  <html lang="en">  <head>  <meta charset="UTF-8" />  <meta http-equiv="X-UA-Compatible" content="IE=edge" />  <meta name="viewport" content="width=device-width, initial-scale=1.0" />  <link href="https://cdn.bootcdn.net/ajax/libs/twitter-bootstrap/5.0.2/css/bootstrap.min.css" rel="stylesheet" />  <title>通过set自动更新dom</title>  </head>  <body>  <div class="card" style="width: 300px; margin: 100px auto">  <div class="card-body">  <h1 id="name"></h1>  <button id="btn" class="btn btn-primary">修改</button>  </div>  </div>  <script>  // 获取DOM节点  const name = document.getElementById('name')  const btn = document.getElementById('btn')  // 定义一个修改值的函数  const updateDOM = (el, value) => {  el.innerHTML = value  }  const person = new Proxy({  name: '一碗粥',  }, {  set(target, propKey, value) {  // 如果里面的值改变就去调用我们的updateDOM  updateDOM(name, value)  target[propKey] = value  return true  },  })  name.innerHTML = person.name  // 点击按钮触发修改操作  btn.addEventListener('click', () => {  person.name === '一碗周' ?  (person.name = '一碗粥') :  (person.name = '一碗周')  })  </script>  </body>  </html>  **值得注意的是** 如果目标对象自身的某个属性不可写，那么set方法将不起作用。  **apply()**  apply方法拦截函数的调用、call、apply和Reflect.apply()操作。  该方法接受三个参数分别是目标对象、目标对象的上下文对象（this）和目标对象的参数数组。  示例代码如下：  const proxy = new Proxy(sayMe = () => { }, {  apply (target, thisArg, args) {  return '一碗周'  }  })  console.log(proxy()) // 一碗周  值得注意的是目标对象必须是可调用的，也就是说必须是一个函数。  **construct()**  construct用于拦截new关键字或者Reflect.construct()。  该方法接受三个参数，分别是目标对象、constructor的参数列表和最初被调用的构造函数。该方法必须返回一个对象，否则会抛出异常。  示例代码如下：  function Hero (name) {  this.name = name  }  const proxy = new Proxy(Hero, {  construct (target, argArray, newTarget) {  console.log('construct执行了');  return new target(...argArray)  }  })  new proxy('一碗周') // construct执行了  值得注意的是：由于construct()拦截的是构造函数，所以它的目标对象必须是函数，否则就会报错。  **has()**  has方法主要是用来拦截in操作符的，该方法接受两个参数，分别指的是目标对象和需要查询的属性名。  我们通过has方法，使\_开头的属性不被in运算符发现，实现代码如下：  const person = {  name: '一碗周',  \_hobby: 'coding'  }  const proxy = new Proxy(person, {  /\*\*  \* @param {object} target 目标对象  \* @param {string} propKey 需查询的属性名  \* @returns {boolean}  \*/  has (target, propKey) {  if (propKey[0] === '\_') return false  return propKey in target  }  })  console.log('name' in proxy) // true  console.log('\_hobby' in proxy) // false  值得注意的是如果原对象不可配置或者禁止扩展，这时has就会抛出异常。  **deleteProperty()**  deleteProperty方法用于拦截对属性执行的delete操作(delete关键字以及Reflect.deleteProperty()方法)，如果返回false或者抛出异常则无法对属性执行删除操作。  该方法接受两个参数，分别是是目标对象和被删除的属性名。  现在我们需要对\_开头的属性不可进行delete操作，示例代码如下：  const person = {  name: '一碗周',  \_hobby: 'coding'  }  // 不可对 \_ 开头的属性进行删除  const proxy = new Proxy(person, {  /\*\*  \* @param {object} target 目标对象  \* @param {string} propKey 需删除的属性名  \* @returns {boolean} 返回true表示删除成功，返回false不可删除  \*/  deleteProperty (target, propKey) {  if (propKey[0] === '\_') return false  return true  }  })  delete proxy.\_hobby  console.log(proxy.\_hobby) // coding  **值得注意的是** 如果目标对象的属性是不可配置的，那么该属性不能被删除。  **getPrototypeOf()**  getPrototypeOf会拦截如下五种操作（方法/属性/运算符）：   * Object.prototype.\_\_proto\_\_ * Object.prototype.isPrototypeOf() * Object.getPrototypeOf()<br /> * Reflect.getPrototypeOf() * instanceof   该方法接受一个参数，就是被代理的目标对象，返回值是一个对象或者null。  示例代码如下：  var obj = {};  var p = new Proxy(obj, {  getPrototypeOf (target) {  return Array.prototype;  }  });  console.log(  Object.getPrototypeOf(p) === Array.prototype, // true  Reflect.getPrototypeOf(p) === Array.prototype, // true  p.\_\_proto\_\_ === Array.prototype, // true  Array.prototype.isPrototypeOf(p), // true  p instanceof Array // true  );  **setPrototypeOf()**  setPrototypeOf会拦截如下两种种操作：   * Object.setPrototypeOf() * Reflect.setPrototypeOf()   该方法接受两个参数，一个是**要被修改的原型的目标对象** 和**对象的新原型或者null** 。方法返回一个布尔值，返回true表示修改成功，返回false表示修改失败。  示例代码如下(只要修改原型就会抛出异常)：  var proto = {};  var proxy = new Proxy({}, {  /\*\*  \* @param {object} target 要被修改的原型的目标对象  \* @param {object} proto 对象的新原型或者null  \* @returns {boolean} true表示修改成功，返回false表示修改失败  \*/  setPrototypeOf (target, proto) {  throw new Error('无法改变原型');  }  });  Object.setPrototypeOf(proxy, proto);  // Error: 无法改变原型  **isExtensible()**  isExtensible方法用于拦截Object.isExtensible()或者Reflect.isExtensible()操作。该方法接受一个参数，就是目标对象，返回一个布尔值(否则就会抛出异常)。  示例代码如下：  const proxy = new Proxy({}, {  isExtensible () {  console.log('isExtensible执行了');  return true  }  })  Object.isExtensible(proxy) // isExtensible执行了  **preventExtensions()**  preventExtensions方法用于拦截Object.preventExtensions()或者Reflect.preventExtensions()操作。该方法接受一个参数，就是目标对象，返回一个布尔值(如果目标对象是可扩展的，那么只能返回 false)。  示例代码如下：  const proxy = new Proxy({}, {  preventExtensions (target) {  console.log('preventExtensions执行了')  Object.preventExtensions(target)  return true  }  })  Object.preventExtensions(proxy) // preventExtensions执行了  **getOwnPropertyDescriptor()**  getOwnPropertyDescriptor方法用于拦截Object.getOwnPropertyDescriptor()或者Reflect.getOwnPropertyDescriptor()操作。  该方法接受两个参数，一个是**目标对象** 和**返回属性名称的描述** 。该方法的返回值必须是一个Object或者undefined。  示例代码如下：  const proxy = new Proxy({  name: '一碗周'  }, {  /\*\*  \*  \* @param {object} target 目标对象  \* @param {string} propKey 属性名  \*/  getOwnPropertyDescriptor (target, propKey) {  console.log('getOwnPropertyDescriptor执行了');  return Object.getOwnPropertyDescriptor(target, propKey)  }  })  Object.getOwnPropertyDescriptors(proxy, 'name') // getOwnPropertyDescriptor执行了  **defineProperty()**  defineProperty方法用于拦截以下操作：   * Object.defineProperty() * Reflect.defineProperty() * proxy.property='value'   该方法接受三个参数，分别是目标对象，属性名和待定义或修改的属性的描述符。返回值是一个布尔型表示是否修改成功。  示例代码如下：  const proxy = new Proxy({  name: '一碗周'  }, {  /\*\*  \*  \* @param {object} target 目标对象  \* @param {string} propKey 属性名  \* @param {object} descriptor 待定义或修改的属性的描述符  \* @returns {boolean} 是否修改成功  \*/  defineProperty (target, propKey, descriptor) {  console.log('defineProperty执行了');  Object.defineProperty(target, propKey, descriptor)  return true  }  })  Object.defineProperty(proxy, 'name', {  value: '一碗周',  writable: false  }) // defineProperty执行了  console.log(proxy) // { name: '一碗周' }  **ownKeys()**  ownKeys方法用来拦截对象自身属性的读取操作。具体来说，拦截以下操作。   * Object.getOwnPropertyNames()<br /> * Object.getOwnPropertySymbols()<br /> * Object.keys() * Reflect.ownKeys() * for...in循环   该方法接受一个参数，就是目标对象，必须返回一个可枚举对象，否则就会抛出异常。  示例代码如下：  const proxy = new Proxy({}, {  ownKeys: function (target) {  console.log('ownKeys执行了');  return ['a', 'b', 'c'];  }  });  Object.getOwnPropertyNames(proxy) // ownKeys执行了  **Proxy.revocable()**  Proxy.revocable() 方法可以用来创建一个可撤销的拦截对象。  语法结构如下所示：  let { proxy, revoke } = Proxy.revocable(target, handler);   * proxy：拦截对象，与new Proxy(target, handler)创建的无区别 * revoke：一个方法，执行该方法及撤销这个拦截对象 * target和handler同上   示例代码如下：  const person = {  name: '一碗周'  }  const { proxy, revoke } = Proxy.revocable(person, {  get (target, propKey, receiver) {  if (propKey in target) {  return target[propKey]  } else {  throw new ReferenceError(`属性：${propKey} 不存在`)  }  }  })  console.log(proxy.name)  // console.log(proxy.age) // ReferenceError: 属性：age 不存在  // 调用revoke无需传递任何参数  revoke()  // 一旦撤销就不允许再次访问  console.log(proxy.name) // Cannot perform 'get' on a proxy that has been revoked  **this指向**  在Proxy中参数target中的this是指向target的，而handler中的this是指向handler的，具体参考  <https://es6.ruanyifeng.com/#docs/proxy#this> |

## Proxy参考文档2

|  |
| --- |
| 在支持 Proxy 的浏览器环境中，Proxy 是一个全局对象，可以直接使用。Proxy(target, handler) 是一个构造函数，target 是被代理的对象，handlder 是声明了各类代理操作的对象，最终返回一个代理对象。外界每次通过代理对象访问 target 对象的属性时，就会经过 handler 对象，从这个流程来看，代理对象很类似 middleware（中间件）。那么 Proxy 可以拦截什么操作呢？最常见的就是 get（读取）、set（修改）对象属性等操作，完整的可拦截操作列表请点击[这里](http://www.ecma-international.org/ecma-262/6.0/#sec-proxy-object-internal-methods-and-internal-slots)。此外，Proxy 对象还提供了一个 revoke 方法，可以随时注销所有的代理操作。在我们正式介绍 Proxy 之前，建议你对 Reflect 有一定的了解，它也是一个 ES6 新增的全局对象，详细信息请参考 [MDN Reflect](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Reference/Global_Objects/Reflect)。 Basic const target = {  name: 'Billy Bob',  age: 15  };  const handler = {  get(target, key, proxy) {  const today = new Date();  console.log(`GET request made for ${key} at ${today}`);  return Reflect.get(target, key, proxy);  }  };  const proxy = new Proxy(target, handler);  proxy.name;  *// => "GET request made for name at Thu Jul 21 2016 15:26:20 GMT+0800 (CST)"*  *// => "Billy Bob"*  在上面的代码中，我们首先定义了一个被代理的目标对象 target，然后声明了包含所有代理操作的 handler 对象，接下来使用 Proxy(target, handler) 创建代理对象 proxy，此后所有使用 proxy 对 target 属性的访问都会经过 handler 的处理。 1. 抽离校验模块 让我们从一个简单的类型校验开始做起，这个示例演示了如何使用 Proxy 保障数据类型的准确性：  let numericDataStore = {  count: 0,  amount: 1234,  total: 14  };  numericDataStore = new Proxy(numericDataStore, {  set(target, key, value, proxy) {  if (typeof value !== 'number') {  throw Error("Properties in numericDataStore can only be numbers");  }  return Reflect.set(target, key, value, proxy);  }  });  *// 抛出错误，因为 "foo" 不是数值*  numericDataStore.count = "foo";  *// 赋值成功*  numericDataStore.count = 333;  如果要直接为对象的所有属性开发一个校验器可能很快就会让代码结构变得臃肿，使用 Proxy 则可以将校验器从核心逻辑分离出来自成一体：  function createValidator(target, validator) {  return new Proxy(target, {  \_validator: validator,  set(target, key, value, proxy) {  if (target.hasOwnProperty(key)) {  let validator = this.\_validator[key];  if (!!validator(value)) {  return Reflect.set(target, key, value, proxy);  } else {  throw Error(`Cannot set ${key} to ${value}. Invalid.`);  }  } else {  throw Error(`${key} is not a valid property`)  }  }  });  }  const personValidators = {  name(val) {  return typeof val === 'string';  },  age(val) {  return typeof age === 'number' && age > 18;  }  }  class Person {  constructor(name, age) {  this.name = name;  this.age = age;  return createValidator(this, personValidators);  }  }  const bill = new Person('Bill', 25);  *// 以下操作都会报错*  bill.name = 0;  bill.age = 'Bill';  bill.age = 15;  通过校验器和主逻辑的分离，你可以无限扩展 personValidators 校验器的内容，而不会对相关的类或函数造成直接破坏。更复杂一点，我们还可以使用 Proxy 模拟类型检查，检查函数是否接收了类型和数量都正确的参数：  let obj = {  pickyMethodOne: function(obj, str, num) { */\* ... \*/* },  pickyMethodTwo: function(num, obj) { */\*... \*/* }  };  const argTypes = {  pickyMethodOne: ["object", "string", "number"],  pickyMethodTwo: ["number", "object"]  };  obj = new Proxy(obj, {  get: function(target, key, proxy) {  var value = target[key];  return function(...args) {  var checkArgs = argChecker(key, args, argTypes[key]);  return Reflect.apply(value, target, args);  };  }  });  function argChecker(name, args, checkers) {  for (var idx = 0; idx < args.length; idx++) {  var arg = args[idx];  var type = checkers[idx];  if (!arg || typeof arg !== type) {  console.warn(`You are incorrectly implementing the signature of ${name}. Check param ${idx + 1}`);  }  }  }  obj.pickyMethodOne();  *// > You are incorrectly implementing the signature of pickyMethodOne. Check param 1*  *// > You are incorrectly implementing the signature of pickyMethodOne. Check param 2*  *// > You are incorrectly implementing the signature of pickyMethodOne. Check param 3*  obj.pickyMethodTwo("wopdopadoo", {});  *// > You are incorrectly implementing the signature of pickyMethodTwo. Check param 1*  *// No warnings logged*  obj.pickyMethodOne({}, "a little string", 123);  obj.pickyMethodOne(123, {}); 2. 私有属性 在 JavaScript 或其他语言中，大家会约定俗成地在变量名之前添加下划线 \_ 来表明这是一个私有属性（并不是真正的私有），但我们无法保证真的没人会去访问或修改它。在下面的代码中，我们声明了一个私有的 apiKey，便于 api 这个对象内部的方法调用，但不希望从外部也能够访问 api.\_apiKey:  var api = {  \_apiKey: '123abc456def',  */\* mock methods that use this.\_apiKey \*/*  getUsers: function(){},  getUser: function(userId){},  setUser: function(userId, config){}  };  *// logs '123abc456def';*  console.log("An apiKey we want to keep private", api.\_apiKey);  *// get and mutate \_apiKeys as desired*  var apiKey = api.\_apiKey;  api.\_apiKey = '987654321';  很显然，约定俗成是没有束缚力的。使用 ES6 Proxy 我们就可以实现真实的私有变量了，下面针对不同的读取方式演示两个不同的私有化方法。第一种方法是使用 set / get 拦截读写请求并返回 undefined:  let api = {  \_apiKey: '123abc456def',  getUsers: function(){ },  getUser: function(userId){ },  setUser: function(userId, config){ }  };  const RESTRICTED = ['\_apiKey'];  api = new Proxy(api, {  get(target, key, proxy) {  if(RESTRICTED.indexOf(key) > -1) {  throw Error(`${key} is restricted. Please see api documentation for further info.`);  }  return Reflect.get(target, key, proxy);  },  set(target, key, value, proxy) {  if(RESTRICTED.indexOf(key) > -1) {  throw Error(`${key} is restricted. Please see api documentation for further info.`);  }  return Reflect.get(target, key, value, proxy);  }  });  *// 以下操作都会抛出错误*  console.log(api.\_apiKey);  api.\_apiKey = '987654321';  第二种方法是使用 has 拦截 in 操作：  var api = {  \_apiKey: '123abc456def',  getUsers: function(){ },  getUser: function(userId){ },  setUser: function(userId, config){ }  };  const RESTRICTED = ['\_apiKey'];  api = new Proxy(api, {  has(target, key) {  return (RESTRICTED.indexOf(key) > -1) ?  false :  Reflect.has(target, key);  }  });  *// these log false, and `for in` iterators will ignore \_apiKey*  console.log("\_apiKey" in api);  for (var key in api) {  if (api.hasOwnProperty(key) && key === "\_apiKey") {  console.log("This will never be logged because the proxy obscures \_apiKey...")  }  } 3. 访问日志 对于那些调用频繁、运行缓慢或占用执行环境资源较多的属性或接口，开发者会希望记录它们的使用情况或性能表现，这个时候就可以使用 Proxy 充当中间件的角色，轻而易举实现日志功能：  let api = {  \_apiKey: '123abc456def',  getUsers: function() { */\* ... \*/* },  getUser: function(userId) { */\* ... \*/* },  setUser: function(userId, config) { */\* ... \*/* }  };  function logMethodAsync(timestamp, method) {  setTimeout(function() {  console.log(`${timestamp} - Logging ${method} request asynchronously.`);  }, 0)  }  api = new Proxy(api, {  get: function(target, key, proxy) {  var value = target[key];  return function(...arguments) {  logMethodAsync(new Date(), key);  return Reflect.apply(value, target, arguments);  };  }  });  api.getUsers(); 4. 预警和拦截 假设你不想让其他开发者删除 noDelete 属性，还想让调用 oldMethod 的开发者了解到这个方法已经被废弃了，或者告诉开发者不要修改 doNotChange 属性，那么就可以使用 Proxy 来实现：  let dataStore = {  noDelete: 1235,  oldMethod: function() {*/\*...\*/* },  doNotChange: "tried and true"  };  const NODELETE = ['noDelete'];  const NOCHANGE = ['doNotChange'];  const DEPRECATED = ['oldMethod'];  dataStore = new Proxy(dataStore, {  set(target, key, value, proxy) {  if (NOCHANGE.includes(key)) {  throw Error(`Error! ${key} is immutable.`);  }  return Reflect.set(target, key, value, proxy);  },  deleteProperty(target, key) {  if (NODELETE.includes(key)) {  throw Error(`Error! ${key} cannot be deleted.`);  }  return Reflect.deleteProperty(target, key);  },  get(target, key, proxy) {  if (DEPRECATED.includes(key)) {  console.warn(`Warning! ${key} is deprecated.`);  }  var val = target[key];  return typeof val === 'function' ?  function(...args) {  Reflect.apply(target[key], target, args);  } :  val;  }  });  *// these will throw errors or log warnings, respectively*  dataStore.doNotChange = "foo";  delete dataStore.noDelete;  dataStore.oldMethod(); 5. 过滤操作 某些操作会非常占用资源，比如传输大文件，这个时候如果文件已经在分块发送了，就不需要在对新的请求作出相应（非绝对），这个时候就可以使用 Proxy 对当请求进行特征检测，并根据特征过滤出哪些是不需要响应的，哪些是需要响应的。下面的代码简单演示了过滤特征的方式，并不是完整代码，相信大家会理解其中的妙处：  let obj = {  getGiantFile: function(fileId) {*/\*...\*/* }  };  obj = new Proxy(obj, {  get(target, key, proxy) {  return function(...args) {  const id = args[0];  let isEnroute = checkEnroute(id);  let isDownloading = checkStatus(id);  let cached = getCached(id);  if (isEnroute || isDownloading) {  return false;  }  if (cached) {  return cached;  }  return Reflect.apply(target[key], target, args);  }  }  }); 6. 中断代理 Proxy 支持随时取消对 target 的代理，这一操作常用于完全封闭对数据或接口的访问。在下面的示例中，我们使用了 Proxy.revocable 方法创建了可撤销代理的代理对象：  let sensitiveData = { username: 'devbryce' };  const {sensitiveData, revokeAccess} = Proxy.revocable(sensitiveData, handler);  function handleSuspectedHack(){  revokeAccess();  }  *// logs 'devbryce'*  console.log(sensitiveData.username);  handleSuspectedHack();  *// TypeError: Revoked*  console.log(sensitiveData.username); Decorator ES7 中实现的 Decorator，相当于设计模式中的装饰器模式。如果简单地区分 Proxy 和 Decorator 的使用场景，可以概括为：Proxy 的核心作用是控制外界对被代理者内部的访问，Decorator 的核心作用是增强被装饰者的功能。只要在它们核心的使用场景上做好区别，那么像是访问日志这样的功能，虽然本文使用了 Proxy 实现，但也可以使用 Decorator 实现，开发者可以根据项目的需求、团队的规范、自己的偏好自由选择。  原文: <https://www.w3cplus.com/javascript/use-cases-for-es6-proxies.html> © [w3cplus.com](https://www.w3cplus.com/) |

# 15.es6中的Reflect

|  |
| --- |
|  |
|  |

## 1.Reflect对象可以代替Object的某些方法，如Object的defineProperty方法

|  |
| --- |
| //Reflect可以代替Object的某些方法          let obj = {}          Object.defineProperty(obj,"name",{              value:"Kenny",              writable:false, //表明改属性是否可以修改，false表示不可修改              enumerable:false //表明该属性是否可以配置，false表示不可以，此时这个属性不能删除          }) |
|  |

## 我们把它改为Reflect

|  |
| --- |
| //Reflect可以代替Object的某些方法          let obj = {}          // Object.defineProperty(obj,"name",{          //     value:"Kenny",          //     writable:false, //表明改属性是否可以修改，false表示不可修改          //     enumerable:false //表明该属性是否可以配置，false表示不可以，此时这个属性不能删除          // })          Reflect.defineProperty(obj,"name",{              value:"Kenny",              writable:false, //表明改属性是否可以修改，false表示不可修改              enumerable:false //表明该属性是否可以配置，false表示不可以，此时这个属性不能删除          }) |
|  |

### 效果是一样的，但是在以后如果我们需要给Object添加方法我们都应该使用Reflect，我们以后都要从Reflect中去拿Object的内部方法。

## 2.Reflect的第二个用法，用if..else来代替try..catch

|  |
| --- |
|  |
|  |

## 课堂演示，

### 当对象的一个属性默认是可写和可配置的，可以定义多次这个相同的属性，后面的值会覆盖前面的值

|  |
| --- |
|  |
|  |

### 然后我们把那两个属性的一个或者全部改为false，此时将同一个属性定义2次就会抛异常

|  |
| --- |
|  |
|  |

### 然后我们改为Reflect试一试

|  |
| --- |
|  |
|  |

#### 它只是 返回false，没有抛异常

## 3.Reflect的第三给用法，可以将一些in ，delete等等的操作符改为使用函数来处理

|  |
| --- |
|  |

### Reflect有一个set方法可以设置对象的属性

|  |
| --- |
|  |

### 注意：给对象添加属性时，如果使用字面量和obj.属性名=属性值的方法添加的属性和用Reflect.set方法设置的属性都可以删除，但是使Reflect.defineProperty(obj,"name",{value:"jack"})添加的属性无法删除

|  |
| --- |
| let obj ={}          // obj.name="Mary" //用这种方法定义的属性可以删除          // obj.age=30          // Reflect.defineProperty(obj,"name",{value:"jack"})          // Reflect.defineProperty(obj,"age",{value:30}) //用这种方法定义的属性无法删除          Reflect.deleteProperty(obj,"age")          console.log(obj);          // let obj2 ={name:"Betty",age:36,gender:"female"} //字面量属性可以删除          // Reflect.deleteProperty(obj2,"gender")          // console.log(obj2); |
| //用Reflect的set方法来设置对象的属性          let obj3 ={}          Reflect.set(obj3,"name","kenny") //用这种方法设置的属性也可以删除          Reflect.set(obj3,"age",1000)          console.log(obj3);          Reflect.deleteProperty(obj3,"name")          console.log(obj3); |
|  |

### Reflect也有一个get方法可以获取属性

## 4、Reflect对象的用法4：Reflect对象可以和Proxy对象搭配使用

|  |
| --- |
|  |

## 5.Reflect配合Proxy拦截数组，其实非常简单就是利用Reflect获取他的默认行为进行调用

|  |
| --- |
| let arr = [1,2,4]          let arrProxy = new Proxy(arr,{              get(){                console.log("get called....");                return Reflect.get(...arguments)              },              set(target,key,value){                  console.log("set called...");                return Reflect.set(...arguments)              }          }) |
|  |

# 16.es6Promise对象

## 1. Promise 状态

### 状态的特点

Promise 异步操作有三种状态：pending（进行中）、fulfilled（已成功）和 rejected（已失败）。除了异步操作的结果，任何其他操作都无法改变这个状态。

Promise 对象只有：从 pending 变为 fulfilled 和从 pending 变为 rejected 的状态改变。只要处于 fulfilled 和 rejected ，状态就不会再变了即 resolved（已定型）。

### Promise状态的缺点

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）。

## then 方法

then 方法接收两个函数作为参数，第一个参数是 Promise 执行成功时的回调，第二个参数是 Promise 执行失败时的回调，两个函数只会有一个被调用。

### then 方法的特点

在 JavaScript 事件队列的当前运行完成之前，回调函数永远不会被调用。

### 课堂笔记

## Promise基本使用

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

### 可以传递参数

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

|  |
| --- |
|  |
|  |

### 如果在Promise里面把resolve和reject函数都调用，那也只会有一种结果要么成功，要么失败，状态只会改变一次，后续就不会在改变

|  |
| --- |
|  |
|  |

#### 上面的例子中先调用了resolve返回成功的状态，Promise就由pending状态变为fulfilled状态，然后reject的代码就不会执行。

### 课堂案例，封装简单版本的ajax请求

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

### Promise的链式调用的问题

### 把我们上面的简单Promise的代码改为链式调用

|  |
| --- |
|  |
|  |

#### 为什么是undefined，因为第一个then函数里面没有返回内容，那么第二个then自然就拿不到数据了

### 我们修改一下代码，在第一个then里面由把res返回，

|  |
| --- |
|  |
|  |

#### 那么 ，第二个then函数里面也能拿到数据

#### 注意：如果第一个then返回一个不是Promise类型的数据，Promise的状态就只能是pending-》fulfilled，如果它返回 一个Promise对象，那么Promise的状态就由这个Promise对象的结果来决定

### Promise.all([])方法，参数是一个Promise对象数组，只有当这个数组的索引Promise对象执行完成，这个方法才会返回，他会将所有Promise对象的结果一一对应打包到一个数组中作为返回值，如

|  |
| --- |
| let p1 = new Promise(function (resolve,reject) {         setTimeout(()=>{            resolve(10000)         },1000)      })     let p2 = new Promise(function (resolve,reject) {         setTimeout(()=>{            resolve(20000)         },2000)      })     let p3 = new Promise(function (resolve,reject) {         setTimeout(()=>{            resolve(30000)         },3000)      })        Promise.all([p1,p2,p3])         .then(res =>console.log(res))         .catch(err =>console.log(err)) |
|  |

### 但是，如果我们把第二个promise的结果改为失败，就只会返回一个失败的结果

|  |
| --- |
| //Promise.all([])   //     let p1 = new Promise(function (resolve,reject) {         setTimeout(()=>{            resolve(10000)         },1000)      })     let p2 = new Promise(function (resolve,reject) {         setTimeout(()=>{          //   resolve(20000)            reject("connection time out!!!")         },2000)      })     let p3 = new Promise(function (resolve,reject) {         setTimeout(()=>{            resolve(30000)         },3000)      })        Promise.all([p1,p2,p3])         .then(res =>console.log(res))         .catch(err =>console.log(err)) |
|  |

### 即使你把3个promise的结果都给改为失败，它也只有一个失败的结果

|  |
| --- |
| //Promise.all([])   //     let p1 = new Promise(function (resolve,reject) {         setTimeout(()=>{          reject("connection time out1!!!")          //   resolve(10000)         },1000)      })     let p2 = new Promise(function (resolve,reject) {         setTimeout(()=>{          //   resolve(20000)            reject("connection time out2!!!")         },2000)      })     let p3 = new Promise(function (resolve,reject) {         setTimeout(()=>{          reject("connection time out3!!!")          //   resolve(30000)         },3000)      })        Promise.all([p1,p2,p3])         .then(res =>console.log(res))         .catch(err =>console.log(err)) |
|  |

#### 说明只要有一个promise执行失败， all方法就返回reject

### Promise.race([])方法，也是以一个数组作为参数，只要有一个promise对象有结果，race方法就以这个promise的结果作为结果

|  |
| --- |
| let p1 = new Promise(function (resolve,reject) {         setTimeout(()=>{          // reject("connection time out1!!!")            resolve(10000)         },1000)      })     let p2 = new Promise(function (resolve,reject) {         setTimeout(()=>{            resolve(20000)          //   reject("connection time out2!!!")         },2000)      })     let p3 = new Promise(function (resolve,reject) {         setTimeout(()=>{          // reject("connection time out3!!!")            resolve(30000)         },3000)      })          //Promise.race()方法      Promise.race([p1,p2,p3])         .then(res =>console.log(res))         .catch(err =>console.log(err)) |
|  |
| let p1 = new Promise(function (resolve,reject) {         setTimeout(()=>{          // reject("connection time out1!!!")            resolve(10000)         },1000)      })     let p2 = new Promise(function (resolve,reject) {         setTimeout(()=>{          //   resolve(20000)            reject("connection time out2!!!")         },200)      })     let p3 = new Promise(function (resolve,reject) {         setTimeout(()=>{          // reject("connection time out3!!!")            resolve(30000)         },3000)      })          //Promise.race()方法      Promise.race([p1,p2,p3])         .then(res =>console.log(res))         .catch(err =>console.log(err))     </script> |
|  |
| let p1 = new Promise(function (resolve,reject) {         setTimeout(()=>{          // reject("connection time out1!!!")            resolve(10000)         },1000)      })     let p2 = new Promise(function (resolve,reject) {         setTimeout(()=>{            resolve(20000)          //   reject("connection time out2!!!")         },2000)      })     let p3 = new Promise(function (resolve,reject) {         setTimeout(()=>{          // reject("connection time out3!!!")            resolve(30000)         },300)      })          //Promise.race()方法      Promise.race([p1,p2,p3])         .then(res =>console.log(res))         .catch(err =>console.log(err)) |
|  |

## Promise参考文档1

|  |
| --- |
| **Promise 对象有以下两个特点:**  **1、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: 初始状态，不是成功或失败状态。**  **fulfilled: 意味着操作成功完成。**  **rejected: 意味着操作失败。**  **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，它的英语意思就是「承诺」，表示其他手段无法改变。**  **2、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 Resolved 和从 Pending 变为 Rejected。只要这两种情况发生，状态就凝固了，不会再变了，会一直保持这个结果。就算改变已经发生了，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 优缺点**  **有了 Promise 对象，就可以将异步操作以同步操作的流程表达出来，避免了层层嵌套的回调函数。此外，Promise 对象提供统一的接口，使得控制异步操作更加容易。**  **Promise 也有一些缺点。首先，无法取消 Promise，一旦新建它就会立即执行，无法中途取消。其次，如果不设置回调函数，Promise 内部抛出的错误，不会反应到外部。第三，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）。**  **Promise 创建**  **要想创建一个 promise 对象、可以使用 new 来调用 Promise 的构造器来进行实例化。**  **下面是创建 promise 的步骤：**  **var promise = new Promise(function(resolve, reject) {**  **// 异步处理**  **// 处理结束后、调用resolve 或 reject**  **});**  **Promise 构造函数包含一个参数和一个带有 resolve（解析）和 reject（拒绝）两个参数的回调。在回调中执行一些操作（例如异步），如果一切都正常，则调用 resolve，否则调用 reject。**  **实例**  **var myFirstPromise = new Promise(function(resolve, reject){**  **//当异步代码执行成功时，我们才会调用resolve(...), 当异步代码失败时就会调用reject(...)**  **//在本例中，我们使用setTimeout(...)来模拟异步代码，实际编码时可能是XHR请求或是HTML5的一些API方法.**  **setTimeout(function(){**  **resolve("成功!"); //代码正常执行！**  **}, 250);**  **});**    **myFirstPromise.then(function(successMessage){**  **//successMessage的值是上面调用resolve(...)方法传入的值.**  **//successMessage参数不一定非要是字符串类型，这里只是举个例子**  **document.write("Yay! " + successMessage);**  **});**  **尝试一下 »**  **对于已经实例化过的 promise 对象可以调用 promise.then() 方法，传递 resolve 和 reject 方法作为回调。**  **promise.then() 是 promise 最为常用的方法。**  **promise.then(onFulfilled, onRejected)**  **promise简化了对error的处理，上面的代码我们也可以这样写：**  **promise.then(onFulfilled).catch(onRejected)**  **Promise Ajax**  **下面是一个用 Promise 对象实现的 Ajax 操作的例子。**  **实例**  **function ajax(URL) {**  **return new Promise(function (resolve, reject) {**  **var req = new XMLHttpRequest();**  **req.open('GET', URL, true);**  **req.onload = function () {**  **if (req.status === 200) {**  **resolve(req.responseText);**  **} else {**  **reject(new Error(req.statusText));**  **}**  **};**  **req.onerror = function () {**  **reject(new Error(req.statusText));**  **};**  **req.send();**  **});**  **}**  **var URL = "/try/ajax/testpromise.php";**  **ajax(URL).then(function onFulfilled(value){**  **document.write('内容是：' + value);**  **}).catch(function onRejected(error){**  **document.write('错误：' + error);**  **});**  **尝试一下 »**  **上面代码中，resolve 方法和 reject 方法调用时，都带有参数。它们的参数会被传递给回调函数。reject 方法的参数通常是 Error 对象的实例，而 resolve 方法的参数除了正常的值以外，还可能是另一个 Promise 实例，比如像下面这样。**  **var p1 = new Promise(function(resolve, reject){**  **// ... some code**  **});**    **var p2 = new Promise(function(resolve, reject){**  **// ... some code**  **resolve(p1);**  **})**  **上面代码中，p1 和 p2 都是 Promise 的实例，但是 p2 的 resolve 方法将 p1 作为参数，这时 p1 的状态就会传递给 p2。如果调用的时候，p1 的状态是 pending，那么 p2 的回调函数就会等待 p1 的状态改变；如果 p1 的状态已经是 fulfilled 或者 rejected，那么 p2 的回调函数将会立刻执行。**  **Promise.prototype.then方法：链式操作**  **Promise.prototype.then 方法返回的是一个新的 Promise 对象，因此可以采用链式写法。**  **getJSON("/posts.json").then(function(json) {**  **return json.post;**  **}).then(function(post) {**  **// proceed**  **});**  **上面的代码使用 then 方法，依次指定了两个回调函数。第一个回调函数完成以后，会将返回结果作为参数，传入第二个回调函数。**  **如果前一个回调函数返回的是Promise对象，这时后一个回调函数就会等待该Promise对象有了运行结果，才会进一步调用。**  **getJSON("/post/1.json").then(function(post) {**  **return getJSON(post.commentURL);**  **}).then(function(comments) {**  **// 对comments进行处理**  **});**  **这种设计使得嵌套的异步操作，可以被很容易得改写，从回调函数的"横向发展"改为"向下发展"。**  **Promise.prototype.catch方法：捕捉错误**  **Promise.prototype.catch 方法是 Promise.prototype.then(null, rejection) 的别名，用于指定发生错误时的回调函数。**  **getJSON("/posts.json").then(function(posts) {**  **// some code**  **}).catch(function(error) {**  **// 处理前一个回调函数运行时发生的错误**  **console.log('发生错误！', error);**  **});**  **Promise 对象的错误具有"冒泡"性质，会一直向后传递，直到被捕获为止。也就是说，错误总是会被下一个 catch 语句捕获。**  **getJSON("/post/1.json").then(function(post) {**  **return getJSON(post.commentURL);**  **}).then(function(comments) {**  **// some code**  **}).catch(function(error) {**  **// 处理前两个回调函数的错误**  **});**  **Promise.all方法，Promise.race方法**  **Promise.all 方法用于将多个 Promise 实例，包装成一个新的 Promise 实例。**  **var p = Promise.all([p1,p2,p3]);**  **上面代码中，Promise.all 方法接受一个数组作为参数，p1、p2、p3 都是 Promise 对象的实例。（Promise.all 方法的参数不一定是数组，但是必须具有 iterator 接口，且返回的每个成员都是 Promise 实例。）**  **p 的状态由 p1、p2、p3 决定，分成两种情况。**  **（1）只有p1、p2、p3的状态都变成fulfilled，p的状态才会变成fulfilled，此时p1、p2、p3的返回值组成一个数组，传递给p的回调函数。**  **（2）只要p1、p2、p3之中有一个被rejected，p的状态就变成rejected，此时第一个被reject的实例的返回值，会传递给p的回调函数。**  **下面是一个具体的例子。**  **// 生成一个Promise对象的数组**  **var promises = [2, 3, 5, 7, 11, 13].map(function(id){**  **return getJSON("/post/" + id + ".json");**  **});**    **Promise.all(promises).then(function(posts) {**  **// ...**  **}).catch(function(reason){**  **// ...**  **});**  **Promise.race 方法同样是将多个 Promise 实例，包装成一个新的 Promise 实例。**  **var p = Promise.race([p1,p2,p3]);**  **上面代码中，只要p1、p2、p3之中有一个实例率先改变状态，p的状态就跟着改变。那个率先改变的Promise实例的返回值，就传递给p的返回值。**  **如果Promise.all方法和Promise.race方法的参数，不是Promise实例，就会先调用下面讲到的Promise.resolve方法，将参数转为Promise实例，再进一步处理。**  **Promise.resolve 方法，Promise.reject 方法**  **有时需要将现有对象转为Promise对象，Promise.resolve方法就起到这个作用。**  **var jsPromise = Promise.resolve($.ajax('/whatever.json'));**  **上面代码将 jQuery 生成 deferred 对象，转为一个新的 ES6 的 Promise 对象。**  **如果 Promise.resolve 方法的参数，不是具有 then 方法的对象（又称 thenable 对象），则返回一个新的 Promise 对象，且它的状态为fulfilled。**  **var p = Promise.resolve('Hello');**    **p.then(function (s){**  **console.log(s)**  **});**  **// Hello**  **上面代码生成一个新的Promise对象的实例p，它的状态为fulfilled，所以回调函数会立即执行，Promise.resolve方法的参数就是回调函数的参数。**  **如果Promise.resolve方法的参数是一个Promise对象的实例，则会被原封不动地返回。**  **Promise.reject(reason)方法也会返回一个新的Promise实例，该实例的状态为rejected。Promise.reject方法的参数reason，会被传递给实例的回调函数。**  **var p = Promise.reject('出错了');**    **p.then(null, function (s){**  **console.log(s)**  **});**  **// 出错了**  **上面代码生成一个Promise对象的实例，状态为rejected，回调函数会立即执行。**  **参考链接：**  **https://wohugb.gitbooks.io/ecmascript-6/content/docs/promise.html**  **https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Reference/Global\_Objects/Promise**  **http://qingtong234.github.io/2016/01/14/javascript%E4%B8%AD%E7%9A%84promise%E5%BC%82%E6%AD%A5%E7%BC%96%E7%A8%8B-1/** |