爱创课堂前端培训

# ECMAScript2015

第2天课堂笔记（本课程共3天）

班级：北京前端训练营23期

讲师：李兰波

日期：2019年7月10日

爱创课堂官网 ：[www.icketang.com](http://www.icketang.com)

# 复习

ES6中的新特点：

let：新增的关键字，用于定义遵循ES6规范的变量。

不可以重复定义

没有声明提升

块级作用域

不会挂载到window身上

for循环中可以记住当时的值

const：新增的关键字，用于定义遵循ES6规范的常量。

不可以重复定义

没有声明提升

块级作用域

不会挂载到window身上

不可以通过=修改对象的值 如果保存的是引用类型则可以通过.语法或者方括号语法进行属性的增加、修改，通过delete关键字，进行属性的删除。

对象的新方法：

Object.is: 用于判定两者是否全等

Object.assign: 用于将第二个参数及后面的每一个参数（每个参数都是对象）所拥有的属性和方法赋予第一个参数。

属于浅复制

字符串：

多行字符串：通过``键定义

字符串模板：${} 括号内是一个JS执行环境，可以使用JS环境中的任何内容以及语法

startsWith：判定一个字符串的指定位置是否是另一个字符串开头

endsWith：判定一个字符串的指定位置是否是另一个字符串结尾

includes：判定一个字符串的指定位置往后是否包含另一个字符串

repeat：将一个字符串复制一定的次数。并返回新的复制后的字符串。

数组：

Array.of：定义数组 每一个参数都是成员

Array.from: 将类数组对象转化为数组

类数组对象：必须具备数字下标与length属性

find：用于模糊查询一个数组中的符合某个特点的成员

参数是一个函数。

该函数返回一个布尔值。该函数会执行数组.length次。每一次如果返回的是false，则继续下一次，如果是true，则中止。并且find方法会返回此时循环到的成员

findIndex:用于模糊查询一个数组中的符合某个特点的成员的索引

参数是一个函数。

该函数返回一个布尔值。该函数会执行数组.length次。每一次如果返回的是false，则继续下一次，如果是true，则中止。并且find方法会返回此时循环到的成员的索引。

copyWithin：用于数组的内部复制

三个参数：

pos： 粘贴的开始位置（包含）

start： 复制的开始位置（包含）

end： 复制的结束位置（不包含）

Symbol：ES6中新增的数据类型。表示一个独一无二的符号。

获取值的方式： Symbol() 参数只有语义的作用

该Symbol类型主要用于解决对象的属性无法重复的问题。

Proxy：代理对象

接收两个参数：

第一个是目标对象

第二个是处理规则对象

get方法：在通过代理对象访问属性时，会执行

参数：

第一个是 目标对象

第二个是 属性名

第三个是 代理对象自己

set方法：在通过代理对象设置属性时，会执行

参数：

第一个是 目标对象

第二个是 属性名

第三个是 属性值

第四个是 代理对象自己

返回值：代理对象

解构语法：

解构对象： let {prop1, prop2, prop3} = obj;

等价代码：

let prop1 = obj.prop1;

let prop2 = obj.prop2;

let prop3 = obj.prop3;

解构数组： let [a, b, c] = arr;

等价代码：

let a = arr[0];

let b = arr[1];

let c = arr[2];

注：解构对象的时候，等号左侧的变量名，顺序无关，属性名称相关。解构数组的时候，等号左侧的变量名，顺序相关。

拓展语法：...语法

获取剩余参数：在定义函数的时候，可以在形参列表中通过...args 获取剩余的所有参数。它之前的参数形参与实参一一对应。

解构数组：在解构数组的时候，如果只想要获取前几个，剩余的数组成员要放入一个新的数组。则可以： let [a, b, ...c] = arr;

等价于：

let a = arr[0];

let b = arr[1];

let c = arr.slice(2);

传递参数：

console.log(arr); => 在控制台中输出的是这个数组

console.log(...arr); => 在控制台中输出的是这个数组的每一个成员

Number的静态方法：

Number.isNaN: 判定一个数字是否是NaN 如果不是数字，则返回false

Number.isFinite: 判定一个数字是否是有限的 如果不是数字，则返回false

Number.isInteger: 判定一个数字是否是整数 如果不是数字，则返回false

箭头函数：

ES6中新增的一种新的函数的定义方式。

语法： () => {}

特点：

this只有在定义函数、声明函数的时候确定。

规则：在定义、声明函数时，所在的作用域中的this。

一旦确定之后，再也无法改变。

函数中添加了一个特点，在定义函数的时候，可以给形参一个默认值。

语法：function fun(形参 = 默认值， 形参1 = 默认值1) {}

# 一、箭头函数

## 1.1 无法作为构造函数

因为ES6中，已经增加了class关键字，所以不需要再使用构造函数去模拟了。而且箭头函数中的this它不变。

demo:

|  |
| --- |
| 1. // 定义一个箭头函数 2. let arrowFun = () => { 3. } 4. new arrowFun(); |

输出：

|  |
| --- |
|  |

## 1.2 省略写法

ES6将代码的省略做到了极致。

规则：

1 如果形参中只有一个参数，则可以省略圆括号

2 如果函数体中只有一条代码而且还是返回值，则可以省略{}和return

demo:

|  |
| --- |
| 1. let fun = a => a + 2; 2. console.log(fun(1)); |

输出结果：

|  |
| --- |
|  |

## 1.3 arguments

箭头函数中，取消了arguments 改为使用拓展语法代替

demo:

|  |
| --- |
| 1. // 箭头函数中，取消了arguments 改为使用拓展语法代替 2. let arrowFun = () => { 3. console.log(arguments); 4. } 5. // 调用 6. arrowFun(); |

报错：

|  |
| --- |
|  |

# 数组的迭代器方法

ES6中，新增了迭代器接口。

## 2.1 keys方法

该方法用于获取数组的所有的keys 也就是下标、索引

demo1: 获取迭代器对象

|  |
| --- |
| 1. // 定义数组 2. let arr = ["a", "b", "c", "d", "e"]; 3. // 调用keys方法 返回迭代器对象 4. let iterator = arr.keys(); 5. console.log(iterator); |

输出1：

|  |
| --- |
|  |

demo2: 迭代器对象调用一次next

|  |
| --- |
| 1. // 每一次迭代器对象调用next方法 都会返回一次结果 2. let result1 = iterator.next(); 3. console.log(result1); |

输出2：

|  |
| --- |
|  |

本次的输出结果是一个对象，对象中有value属性，就是我们这一次得到的结果。对象中有done属性，表示迭代是否完成。false表示未完成，就可以继续。如果true，表示已经完成。

demo3: 代码多次调用next

|  |
| --- |
| 1. let result1 = iterator.next(); 2. console.log(result1); 3. console.log(iterator.next()); 4. console.log(iterator.next()); 5. console.log(iterator.next()); 6. console.log(iterator.next()); 7. console.log(iterator.next()); 8. console.log(iterator.next()); 9. console.log(iterator.next()); |

输出3：

|  |
| --- |
|  |

## 2.2 values方法

该方法与keys方法的使用方式一致，区别：

keys方法的迭代器对象调用了next之后返回的对象中的value是下标

values方法的迭代器对象调用了next之后返回的对象中的value是成员

demo:

|  |
| --- |
| 1. // 定义数组 2. let arr = ["a", "b", "c", "d", "e"]; 3. // 调用values方法 返回迭代器对象 4. let iterator = arr.values(); 5. // 每一次迭代器对象调用next方法 都会返回一次结果 6. let result1 = iterator.next(); 7. console.log(result1); 8. console.log(iterator.next()); 9. console.log(iterator.next()); 10. console.log(iterator.next()); 11. console.log(iterator.next()); 12. console.log(iterator.next()); 13. console.log(iterator.next()); 14. console.log(iterator.next()); |

输出：

|  |
| --- |
|  |

## 2.3 entries方法

该方法与keys方法、values方法的使用方式一致，区别：

keys方法的迭代器对象调用了next之后返回的对象中的value是下标

values方法的迭代器对象调用了next之后返回的对象中的value是成员

entries方法的迭代器对象调用了next之后返回的对象中的value是长度为2的数组

第一个成员是数组的下标

第二个成员是数组的成员

demo:

|  |
| --- |
| 1. // 定义数组 2. let arr = ["a", "b", "c", "d", "e"]; 3. // 调用entries方法 返回迭代器对象 4. let iterator = arr.entries(); 5. // 每一次迭代器对象调用next方法 都会返回一次结果 6. let result1 = iterator.next(); 7. console.log(result1); 8. console.log(iterator.next()); 9. console.log(iterator.next()); 10. console.log(iterator.next()); 11. console.log(iterator.next()); 12. console.log(iterator.next()); 13. console.log(iterator.next()); 14. console.log(iterator.next()); |

输出：

|  |
| --- |
|  |

# 迭代器

什么叫做迭代器？

定义：给定一种方式，能够顺序的遍历结构内部的数据，同时又不暴露内部结构的方式。

ES6中就定义了一个迭代器接口，并且已经给部分数据结构实现了该迭代器。比如数组。

使用方式：

通常是在数据结构的对象的原型上，定义该迭代器接口的获取方式。在调用了该原型方法之后，会返回迭代器对象。

迭代器对象的使用方式是统一的： iterator.next()

得到的结果的结构也是统一的： {value: xxx, done: boolean}

# for……of循环

该循环是用于循环迭代器、或者实现了迭代器接口的数据结构的。

举例：

调用了数组的keys、values、entries方法之后，得到一个迭代器对象：iterator

for (let i of iterator) {

i: 返回的对象value属性和done属性中的value的值

}

数组是数据结构，数组实现了迭代器接口：所以我们可以通过for of直接迭代数组

let arr = [1, 2, 3, 4, 5, 6];

for (let i of arr) {

i: 成员

}

对象也是数据结构，但是没有实现迭代器接口：所以我们不可以通过for of直接迭代对象

let obj = {};

for (var i of obj) {}

报错：obj is not iterable

demo1: 迭代迭代器接口对象

|  |
| --- |
| 1. // 定义数组 2. let arr = ["a", "b", "c", "d", "e"]; 3. // 调用entries方法 返回迭代器对象 4. let iterator = arr.entries(); 5. // 每一次都手工调用iterator的next 太过繁琐 6. // 所以 ES6提供了for of循环 7. for (let i of iterator) { 8. console.log(i); 9. } |

输出：

|  |
| --- |
|  |

demo2: 迭代实现了迭代器接口的数据结构

|  |
| --- |
| 1. // 直接迭代arr 2. for (let i of arr) { 3. console.log(i); 4. } |

输出2：

|  |
| --- |
|  |

demo3: 迭代一个没有实现迭代器接口的数据结构

|  |
| --- |
| 1. // 定义对象 2. let obj = { 3. a: 1, 4. b: 2, 5. c: 3 6. } 7. // 尝试for of迭代obj 8. for (let i of obj) { 9. } |

输出3：报错

|  |
| --- |
|  |

# 五、新的数据结构

## 5.1 Set

Set是一个新的数据结构，可以认为是一个内容不可重复的数组。

注：初始化的时候，一定要new Set

否则：

demo:

|  |
| --- |
| 1. // 定义新的数据结构 2. let s = new Set([1, 2, 3, 4, 5, 6, 6, 6, 6, 6]); 3. console.log(s); |

输出：

|  |
| --- |
|  |

方法：

add：参数是任意类型的值

作用：将Set容器内，增加一个新的成员。

注：一定不可以添加已有的。如果添加的是已有的成员，添加失败。

delete：参数是任意类型的值

作用：将Set容器内，移除一个指定的成员。 如果该成员存在，则移除。

forEach：参数是函数 ES5中的迭代器方法

作用：根据Set成员的个数，执行函数多次。每一次的函数的参数是Set的某一个成员

函数的三个参数：Set的value、Set的key、Set自身

has：参数是任意的值

作用：判定参数是否已经存在于Set内部

clear: 没有参数。

作用：清空Set内部的所有内容。

## 5.2 WeakSet

WeakSet是一个与Set相关的数据结构。

与Set的区别：

1 每一个成员只能是引用类型

2 WeakSet不影响垃圾回收机制

WeakSet所使用的成员，不会导致标记数值的变化。

demo: WeakSet的成员，会被垃圾回收机制清理掉。

|  |
| --- |
| 1. let ws = new WeakSet([{}, {}]); |

输出：

|  |
| --- |
| 1. Feige图片20190710113508 |

非引用类型成员会报错：

|  |
| --- |
| 1. let ws = new WeakSet([1, 2, 3]); |

报错：

|  |
| --- |
|  |

## 5.3 Map

Map是一个超对象。与对象有关。

普通对象的key只能是字符串。

Map对象的key可以是任意类型的值。可以是字符串、是数字、布尔值、undefined、null、symbol、引用类型值。

初始化：

|  |
| --- |
| 1. let map = new Map(); |

输出map:

|  |
| --- |
|  |

set方法：

接收两个参数

第一个参数是超对象的key。值是任意的。

第二个参数是超对象的value。值也是任意的。

get方法：

接收一个参数

第一个参数是超对象的key。值是任意的

返回值：

该参数key对应的value。

其余方法：

与Set一致。

## 5.4 WeakMap

这是一个弱Map。

与WeakSet一致，WeakSet成员只可以是引用类型。

WeakMap的key，只可以是引用类型。

初始化：

|  |
| --- |
| 1. // 初始化 2. let wm = new WeakMap(); 3. wm.set("a", "1"); |

报错：

与WeakSet一致，WeakSet不会影响到垃圾回收机制的判定。

WeakMap也不会。

|  |
| --- |
|  |

## 5.5 垃圾回收机制

垃圾：内存中没有用的内容，叫做垃圾。比如定义了一个变量，却从来没有使用。

引用计数：

当开辟一个内存地址时，会根据应用它的具体成员的多少来计算数值。当数值不为0时，依旧被使用。当数值为0时，会被清除掉。

标记清除：

进场和出场：整体内存有一个范围，当开辟一个地址时，如果有变量引用，则视为“进场”。如果变量、元素不再引用，则视为“出场”。

引用计数存在一个问题：如果一个元素身上有事件函数，则当把该元素移除时，元素引用函数，函数引用元素。互相引用，导致数字都不为0.无法被移除。所以委托模式，为了解决这个问题，就不给可能被移除的元素自身添加事件。

## 5.6 setInterval与setTimeout

这两个函数，都是用来开启异步代码。

setInterval的参数函数会按照一定的时间间隔执行。

setTimeout的参数函数到时间之后只执行一次。

返回值：

都是数字。

返回值表示的含义是当前的定时器、延时器在浏览器中的编号。

该数字只有在你想要关闭对应的定时器、延时器时才有用。

注：编号是有顺序可循的，所以可以自己猜要关闭哪个。不推荐。推荐的方式还是使用变量来保存这个编号。

# 下午复习

箭头函数：

this的指向规则：当定义这个箭头函数的时候的作用域中的this。就是该箭头函数的this。箭头函数中的this在箭头函数调用的时候是无论如何也不会发生变化的。

不能作为构造函数

获取参数要通过...语法

省略写法：

如果参数只有一个，则可以省略圆括号

如果函数体中只有一条代码并且还是返回值，则可以省略{} 与return关键字

arguments已经不能再使用。

数组的迭代器方法：

keys、values、entries。

这三个方法都会返回迭代器对象。

迭代器对象每调用一次next方法，就会返回一次数据。

格式： {

value：本次要的数据

done: 是否迭代完毕

}

keys的value是数字的下标、索引

values的value是成员值

entries的value是数组:[索引, 成员]

迭代器：

定义：给定一种方式，该方式可以顺序的遍历结构的内部数据，同时又不暴露内部的结构。

for……of循环：

专门用来循环迭代器的语法。

ex: for (let i of iterator) {i: next返回的对象的value属性值}

新的数据结构：

Set：去重数组

Map：超对象 对象的key和value都可以是任意类型

WeakSet：弱Set。特点1：成员只可以是引用类型。特点2：不会影响到垃圾回收机制的判定

WeakMap：弱Map。特点1：对象的key只能是引用类型。特点2：不会影响到垃圾回收机制的判定

垃圾回收机制：

垃圾：内存中已经没有用，但是还占用内存的内容都是垃圾。

第一种：

标记清除 现代高级浏览器使用

划分了内存的有效范围。如果变量保存的地址、或者dom元素。在该范围内，就视为有用。如果地址不再被变量保存，或者是dom元素下树的同时没有被变量保存。此时就视为无用。

第二种：

引用计数 某8使用

针对一个地址的被引用次数而计算它是否有用。

问题：循环引用。 A引用B、同时B也引用A。

setInterval、setTimeout：

它们都是开启一个异步代码。

其实是将函数交给了浏览器，让浏览器等待一段时间之后，将函数推入JS执行队列。

浏览器会返回一个该定时器、延时器的"句柄". 句柄可以理解为“引用”、“钥匙”

想要关闭定时器、延时器的时候，可以通过该“句柄”去关闭。（clearInterval(句柄)、clearTimeout(句柄)）

注：句柄其实是有一定的规律可循的。每开一个延时器、定时器，都会返回一个数字，该数字会累加。所以有经验的程序员可以不通过变量保存句柄。而直接清除指定的定时器、延时器。

但是还是推荐使用变量。

# 六、Promise

Promise是一个构造函数。ES6中新增的。

作用：为了解决异步代码的编程问题。将代码的书写风格，从回调套回调，变为从上到下。

回想一下：Node相册项目中，我们使用的大量的异步编程风格：回调套回调。

这样的话，会导致代码不容易维护。而且代码是横向发展。

所以，为了解决这些问题，ES6中，出现了Promise。

## 6.1 体验Promise

### 6.1.2 在没有Promise的时候发送ajax

|  |
| --- |
| 1. // 发送ajax 2. $.get("/ajax1", function(data) { 3. console.log("第一个ajax发送结束"); 4. // 发送第二个ajax 5. $.get("/ajax2", data, function(data) { 6. console.log("第二个ajax发送结束"); 7. }, "json"); 8. }, "json"); |

红色嵌套了蓝色部分。

这就属于回调函数：又叫做回调地狱。

### 6.1.3 在使用Promise的时候发送ajax

|  |
| --- |
| // 定义第一个任务  let task1 = function() {  return new Promise(function(resolve, reject) {  $.get("/ajax1", function(data) {  resolve(data);  });  });  }   1. // 定义第二个任务 2. let task2 = function(data) { 3. return new Promise(function(resolve, reject) { 4. $.get("/ajax2", data, function(data) { 5. resolve(data); 6. }); 7. }); 8. } 9. // 开始执行 10. let p = task1(); 11. p.then(function(data) { 12. console.log("第一次任务结束"); 13. return task2(data); 14. }) 15. .then(function(data) { 16. console.log("第二次任务结束") 17. }); |

两条代码已经拆分开。谁也不干扰谁。从上往下书写代码。

## 6.2 Promise

Promise是一个构造函数。

接收一个函数A作为参数，通常情况下，会在该函数内发送异步请求。

函数A有两个形参：resolve、reject。这两个形参也是函数。

我们会根据异步的成功还是失败来决定调用resolve还是reject

当调用resolve的时候，会将Promise实例的状态变为resolved。

当调用reject的时候，会将Promise实例的状态变为rejected。

返回值：一个Promise的实例。

第一步：

|  |
| --- |
| // 初始化一个Promise实例  let p = new Promise(function(resolve, reject) {  resolve：函数，当执行的时候会将p的状态变为resolved  reject：函数，当执行的时候会将p的状态变为rejected  }); |

第二步：查看p对象

|  |
| --- |
| 1. // 初始化一个Promise实例 2. let p = new Promise(function(resolve, reject) { 4. }); 5. console.log(p); |

输出：在FireFox浏览器中

|  |
| --- |
|  |

我们观察到：

state属性是pending

有几个原型上的方法：意味着必须要通过实例去调用

catch：用于定义当状态变为rejected时的处理函数

finally：用于定义不管当状态变为什么的时候的处理函数。

then：用于定义当状态变为resolved和状态变为rejected时的处理函数。（接收两个函数作为参数）

此时，我们将Promise的参数中的resolve执行：

|  |
| --- |
| let p = new Promise(function(resolve, reject) {  resolve();  });  console.log(p); |

此时，p的状态变为了: fulfilled 也就是resolved

|  |
| --- |
|  |

再重新将reject执行：

|  |
| --- |
| let p = new Promise(function(resolve, reject) {  // resolve();   1. reject(); 2. }); |

此时，p的状态变为了rejected 并且还报错了。

|  |
| --- |
|  |

报错的原因：因为没有设置resolved时的处理函数。

总结：

Promise接收一个函数作为参数，通常会在该函数体内发送异步请求。并在异步请求结束之后，调用resolve或者reject。

此时Promise实例的状态也会随之变化。

## 6.3 处理函数

当Promise的状态发生变化的时候，我们应当进行后续的逻辑上的代码处理。

“成功时，怎么办？失败时，怎么办？”

Promise的实例上有一个then方法，就是用来处理这个问题的。

promise.then(successCallback, failCallback);

successCallback: 当状态变为resolved时，执行的回调函数

failCallback: 当状态变为rejected时，执行的回调函数

注：promise实例的状态通常跟异步请求的执行结果挂钩

第一种情况：

异步请求成功 => resolve()

异步请求失败 => reject()

但是也可以第二种情况：

异步请求成功 => reject()

异步请求失败 => resolve()

但是没人会执行第二种情况

绑定函数：

|  |
| --- |
| let p = new Promise(function(resolve, reject) {  resolve();  // reject();  });  p.then(function() {  console.log("成功了");  }, function() {  console.log("失败了");  }); |

成功时，也就是resolve()

|  |
| --- |
|  |

失败时，也就是reject()

|  |
| --- |
|  |

如果不执行resolve也不执行reject，则永远不会执行任何代码。

总结：

Promise实例的状态：

只可能  
 pending => resolved/fulfilled

或

pending => rejected

一旦确定状态，则不可能逆向，也不可能改变。

## 6.4 标准的处理异步的方式

这里我们使用setTimeout来模拟异步。

|  |
| --- |
| let p = new Promise(function(resolve, reject) {  // 异步调用  setTimeout(function() {  resolve();  // reject();  }, 3000);  });   1. p.then(function() { 2. console.log("成功了"); 3. }, function() { 4. console.log("失败了") 5. }); |
|  |

## 6.5 传递参数

代码是从异步的回调中抽离到p.then方法的参数函数中了。

但是作用域也变了。从参数函数中，根据作用域是无法得到异步请求回来的数据的。

demo:

|  |
| --- |
| 1. // 第五步 传递参数   let p = new Promise(function(resolve, reject) {  // 发送异步请求  $.ajax({  url: "/ajax1",  dataType: "json",  success: function(data) {  // 判定返回的json数据中的错误代码来决定本次的请求成功还是失败  if (!data.error) {  // 成功  resolve(1, 2, 3, 4, 5);  } else {  // 失败  reject();  }  }  });  });  // 监听状态变化  p.then(function(data) {  console.log(arguments);  // 当前作用域下能够访问的数据 包含自身作用域中定义的内容和全局作用域中的内容  // 访问不到success回调函数中的数据  // 答：作用域已经无法帮助我们获取数据，但是传递参数可以。  // 当reslove函数执行的时候可以传递一个参数。而且仅此一个。  console.log("成功");  }, function() {  console.log("失败");  }); |

输出结果：

|  |
| --- |
|  |

再看reject：

|  |
| --- |
| 1. let p = new Promise(function(resolve, reject) { 2. // 发送异步请求 3. $.ajax({ 4. url: "/ajax1", 5. dataType: "json", 6. success: function(data) { 7. // 判定返回的json数据中的错误代码来决定本次的请求成功还是失败 8. if (!data.error) { 9. // 成功 10. // resolve(1, 2, 3, 4, 5); 11. // 刻意调用失败 12. reject("a", "b", "c", "d"); 13. } else { 14. // 失败 15. // reject(); 16. } 17. } 18. }); 19. }); 20. // 监听状态变化 21. p.then(function(data) { 22. }, function(data) { 23. console.log(data); 24. console.log("失败"); 25. }); |

输出结果：

|  |
| --- |
|  |

结论：

resolve和reject都可以接收一个参数。分别传递给成功处理函数与失败处理函数。

## 6.6 then的连续调用与传递参数

连续调用：

我们可以针对一个Promise实例调用多次then方法。绑定多个处理函数。

demo:

|  |
| --- |
| let p = new Promise(function(resolve, reject) {  $.ajax({  url: "/ajax1",  dataType: "json",  success: function(data) {  // 成功  resolve("A");  }  })  });  // 监听p的状态变化  p.then(function() {  console.log("成功1处理函数");  }, function() {  console.log("失败1处理函数");  })  .then(function() {  console.log("成功2处理函数");  }, function() {  console.log("失败2处理函数");  })  .then(function() {  console.log("成功3处理函数");  }, function() {  console.log("失败3处理函数");  }) |

针对一个Promise实例，绑定了三次then方法。分别绑定了三个“成功处理函数”与“失败处理函数”。

成功时：

|  |
| --- |
|  |

失败时：

|  |
| --- |
|  |

问题：当失败时怎么会调用成功时的处理函数？

解答：

因为第一个then监听的函数执行的时候，p的状态还没有发生变化。所以真的是根据p对象的状态而决定调用哪个处理函数。而当第二个、第三个then监听的函数执行的时候，p的状态已经发生了变化。所以此时只有第一个函数生效。因为它们已经不再监听p的状态的变化了。

结论：

**当针对一个Promise的实例绑定多组处理函数时，只有第一组处理函数，会根据Promise实例的状态而发生变化，后面的每一组，只有第一个函数会执行。**

传递参数：

|  |
| --- |
| let p = new Promise(function(resolve, reject) {  $.ajax({  url: "/ajax1",  dataType: "json",  success: function(data) {  // 成功  // resolve("A");  // 失败  reject("B");  }  })  });  // 监听p的状态变化  p.then(function(data) {  console.log("成功1处理函数", data);  }, function(data) {  console.log("失败1处理函数", data);  })  .then(function(data) {  console.log("成功2处理函数", data);  }, function(data) {  console.log("失败2处理函数", data);  })  .then(function(data) {  console.log("成功3处理函数", data);  }, function(data) {  console.log("失败3处理函数", data);  }) |

此时，只有第一个处理函数会接收到reject传递的参数： 后面的每一个处理函数都没有接收到数据

|  |
| --- |
|  |

因为这种情况下，后面的函数想要获取参数，必须由前一个函数返回

|  |
| --- |
| // 监听p的状态变化  p.then(function(data) {  console.log("成功1处理函数", data);  }, function(data) {  console.log("失败1处理函数", data); // B 这个是reject传递的  return "C";  })  .then(function(data) {  console.log("成功2处理函数", data); // C 这个是前一个处理函数的返回值  return "D";  }, function(data) {  console.log("失败2处理函数", data);  })  .then(function(data) {  console.log("成功3处理函数", data); // D 这个是前一个处理函数的返回值  }, function(data) {  console.log("失败3处理函数", data);  }) |

结论：

**当针对一个Promise实例进行多个函数的处理时，第一个函数的参数，是resolve或者reject传递的。**

**后续的函数中的参数，是由前一个处理函数的return决定的。**

## 6.7 then的连续调用---多个Promise实例

demo

|  |
| --- |
| // 定义一个Promise实例  function task1() {  return new Promise(function(resolve, reject) {  $.get("/ajax1", function(data) {  // 根据返回结果 决定调用resolve reject  resolve();  }, "json");  });  }  function task2() {  return new Promise(function(resolve, reject) {  $.get("/ajax2", function(data) {  // 根据返回结果 决定调用resolve reject  resolve("second");  }, "json");  });  }  let p = task1();  // 监听p的状态变化  p.then(function() {  // 如果成功 我们要发送第二个请求 如果像下面这样书写代码 则又嵌套 所以Promise中规定  // 如果多个Promise连续调用 则在第一个的处理函数中 返回第二个新的Promise实例  // 则后续的then监听的就不再是第一个Promise 而是这个新的Promise实例 后续的then中又会根据第二个Promise  // 的状态变化而决定执行哪个函数  /\*  let p1 = task2();  p1.then()  \*/  console.log("第一个成功返回，发送第二个");  // 正确的写法  return task2();  }, function() {  })  // 下面这个then监听的是task2的返回的Promise实例  .then(function(data) {  console.log("第二个成功", data);  }, function(data) {  console.log("第二个失败", data);  }) |

执行顺序：

1 定义task1

2 定义task2

3 执行task1

返回了一个Promise实例 同时发送/ajax1接口的请求

4 执行完task1 继续向下执行 p变量接收到task1返回的Promise实例 执行then方法 执行第二个then方法

5 所有可执行代码执行完毕

等待第一个接口请求回来 当回来的时候 resolve 所以第一个then方法中的代码执行

return task2();

注：因为返回的是一个新的Promise实例，所以第二个then方法不再监听第一个Promise。改为监听第二个Promise。会根据新Promise的状态而决定执行哪个处理函数

6 task2执行 所以第二个请求/ajax2接口的请求发出去 所有的可执行代码又执行完毕。等待请求回来。

请求回来之后 resolve("second"); 执行

第二个then方法中的第一个参数函数执行。

第二个then方法中的第一个函数接收的数据是新Promise执行resolve() 传递的参数

第二个then方法中的第二个函数接收的数据是新Promise执行reject() 传递的参数

## 6.8 catch

Promise实例的原型上有catch方法。该方法的作用是统一处理所有的前面的Promise实例的rejected状态。并且还可以捕获异常。

注：它只会执行一次。并且会中止后续Promise实例的执行。谁失败，就处理谁。

demo1 : 处理第一个Promise的错误

|  |
| --- |
| 1. function task1() { 2. return new Promise(function(resolve, reject) { 3. // throw new Error("123"); 4. $.get("/ajax1", function(data) { 5. reject("第一个失败"); 6. }, "json"); 7. }); 8. } 9. function task2() { 10. return new Promise(function(resolve, reject) { 11. $.get("/ajax2", function(data) { 12. // 根据返回结果 决定调用resolve reject 13. resolve("second"); 14. // reject("third"); 15. }, "json"); 16. }); 17. } 18. // 初始化第一个任务 19. let p = task1(); 20. // 监听第一个任务 21. p.then(function() { // 此时 每一个Promise实例只需要处理成功的情况 22. // 第一个成功 23. console.log("第一个成功"); 24. return task2(); 25. }) 26. .then(function() { // 此时 每一个Promise实例只需要处理成功的情况 27. // 第二个成功 28. console.log("第二个成功"); 29. }) 30. .catch(function(data) { // 所有的失败的情况都由catch来处理 31. console.log(p); 32. console.log("失败了", data); 33. }); 34. console.log("执行到底"); |

执行结果：

|  |
| --- |
|  |

demo2: 处理第二个失败的情况

|  |
| --- |
| 1. function task1() { 2. return new Promise(function(resolve, reject) { 3. // throw new Error("123"); 4. $.get("/ajax1", function(data) { 5. resolve("第一个成功"); 6. }, "json"); 7. }); 8. } 9. function task2() { 10. return new Promise(function(resolve, reject) { 11. $.get("/ajax2", function(data) { 12. // 根据返回结果 决定调用resolve reject 13. reject("第二个失败"); 14. }, "json"); 15. }); 16. }   // 初始化第一个任务  let p = task1();  // 监听第一个任务  p.then(function() {  // 第一个成功  console.log("第一个成功");  return task2();  })  .then(function() {  // 第二个成功  console.log("第二个成功");  })  .catch(function(data) {  console.log("失败了", data);  });  console.log("执行到底"); |

执行结果：

|  |
| --- |
|  |

demo3: 处理异常

注：它只可以处理Promise所接收的函数中的同步代码的异常

demo3.1 无法处理Promise之外的代码中的异常

|  |
| --- |
| function task1() {  throw new Error("123"); // 这里不属于Promise  return new Promise(function(resolve, reject) {  $.get("/ajax1", function(data) {  resolve("第一个成功");  }, "json");  });  }  function task2() {  return new Promise(function(resolve, reject) {  $.get("/ajax2", function(data) {  // 根据返回结果 决定调用resolve reject  reject("第二个失败");  }, "json");  });  }  // 初始化第一个任务  let p = task1();  // 监听第一个任务  p.then(function() {  // 第一个成功  console.log("第一个成功");  return task2();  })  .then(function() {  // 第二个成功  console.log("第二个成功");  })  .catch(function(data) {  console.log("失败了", data);  });  console.log("执行到底"); |

所以异常被抛给控制台，导致代码无法继续执行

|  |
| --- |
|  |

demo3.2 处理属于Promise的同步代码中的异常

|  |
| --- |
| function task1() {  return new Promise(function(resolve, reject) {  throw new Error("123"); // 这里属于Promise 也是同步代码  $.get("/ajax1", function(data) {  resolve("第一个成功");  }, "json");  });  }   1. function task2() { 2. return new Promise(function(resolve, reject) { 3. $.get("/ajax2", function(data) { 4. // 根据返回结果 决定调用resolve reject 5. reject("第二个失败"); 6. }, "json"); 7. }); 8. } 9. // 初始化第一个任务 10. let p = task1(); 11. // 监听第一个任务 12. p.then(function() { 13. // 第一个成功 14. console.log("第一个成功"); 15. return task2(); 16. }) 17. .then(function() { 18. // 第二个成功 19. console.log("第二个成功"); 20. }) 21. .catch(function(data) { 22. console.log("失败了", data); 23. console.log("虽然失败了，但是还是执行到我了"); 24. }); 25. console.log("执行到底"); |

处理结果：

|  |
| --- |
|  |

demo3.3 无法处理属于Promise的异步代码

|  |
| --- |
| function task1() {  return new Promise(function(resolve, reject) {  $.get("/ajax1", function(data) {  throw new Error("123"); // 这里虽然属于Promise范围内，但是是回调函数中。属于异步，所以catch无法处理  resolve("第一个成功");  }, "json");  });  }   1. function task2() { 2. return new Promise(function(resolve, reject) { 3. $.get("/ajax2", function(data) { 4. // 根据返回结果 决定调用resolve reject 5. reject("第二个失败"); 6. }, "json"); 7. }); 8. } 9. // 初始化第一个任务 10. let p = task1(); 11. // 监听第一个任务 12. p.then(function() { 13. // 第一个成功 14. console.log("第一个成功"); 15. return task2(); 16. }) 17. .then(function() { 18. // 第二个成功 19. console.log("第二个成功"); 20. }) 21. .catch(function(data) { 22. console.log("失败了", data); 23. console.log("虽然失败了，但是还是执行到我了"); 24. }); 25. console.log("执行到底"); |

执行结果：

|  |
| --- |
|  |

## 6.9 finally

finally是Promise原型上的方法。它表示无论如何也要执行的代码。

它接收一个函数作为参数，该函数无论Promise实例的状态如何变化，都会执行。该参数函数没有参数。

demo1: 成功时候的执行

|  |
| --- |
| 1. // 第九步   function task1() {  return new Promise(function(resolve, reject) {  $.get("/ajax1", function(data) {  resolve("第一个成功");  }, "json");  });  }  function task2() {  return new Promise(function(resolve, reject) {  $.get("/ajax2", function(data) {  // 根据返回结果 决定调用resolve reject  resolve("第二个成功");  }, "json");  });  }  // 初始化第一个任务  let p = task1();  // 监听第一个任务  p.then(function() {  // 第一个成功  console.log("第一个成功");  return task2();  })  .then(function() {  // 第二个成功  console.log("第二个成功");  })  .catch(function(data) {  console.log("失败了", data);  console.log("虽然失败了，但是还是执行到我了");  })  .finally(function() {  console.log("最终还是执行了我", arguments);  });  console.log("执行到底"); |

执行结果：

|  |
| --- |
|  |

demo2: 失败时候的执行

|  |
| --- |
| function task1() {  return new Promise(function(resolve, reject) {  $.get("/ajax1", function(data) {  // resolve("第一个成功");  reject("第一个失败");  }, "json");  });  }  function task2() {  return new Promise(function(resolve, reject) {  $.get("/ajax2", function(data) {  // 根据返回结果 决定调用resolve reject  resolve("第二个成功");  }, "json");  });  }  // 初始化第一个任务  let p = task1();  // 监听第一个任务  p.then(function() {  // 第一个成功  console.log("第一个成功");  return task2();  })  .then(function() {  // 第二个成功  console.log("第二个成功");  })  .catch(function(data) {  console.log("失败了", data);  console.log("虽然失败了，但是还是执行到我了");  })  .finally(function() {  console.log("最终还是执行了我", arguments);  });  console.log("执行到底"); |

执行结果：

|  |
| --- |
|  |

demo3: 抛异常时的执行

如果catch没有捕获到该异常，则无法执行finally。

如果catch捕获到异常，等价于rejected。则可以执行finally

## 6.10 补充

then、catch、finally方法随时可以添加。不论Promise实例的状态。不必一定在状态发生变化之前绑定。一般不这么干。

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |