# 复习：

箭头函数：

this指向在定义箭头函数时所处的作用域中的this。

this一旦确定，就再也无法更改。

不能作为构造函数

函数中的参数，可以设置默认值。

函数中的参数，没有arguments，获取剩余参数通过...语法

省略写法：

如果形参只有一个，则可以省略()

如果函数体内只有一条代码而且还是返回值，则省略大括号和return关键字

迭代器：ES6实现了迭代器接口，并且给一部分数据结构对象实现了迭代器接口。

使用规则：

调用迭代器方法：比如数字的keys、values、entries。得到迭代器对象。

该迭代器对象每调用一次next，就会返回一个对象

该对象的结构：value和done

value是本次要遍历的值

done是是否迭代完成

for……of循环语句：

专门用于循环迭代器或者实现了迭代器接口的数据结构的语句。

for (var i of iterator) {

i是 iterator.next().value

}

新的数据结构：

Set：这是一个去重数组。特点：不允许重复的数据存在。（如果是引用类型，则比较地址）

WeakSet：这是一个弱Set。特点：1 成员只可以是引用类型的数据 2 不会影响垃圾回收机制的判定

Map：这是一个超对象。特点：key和value都可以是任意类型。

WeakMap：是一个弱Map。特点：1 key只可以是引用类型的数据 2 不会影响垃圾回收机制的判定

垃圾回收机制：

处理内存中无用变量、数据、内存的机制。

引用计数：每当有一个地址被一个变量引用，则+1。如果一个地址不再被一个变量引用。则-1。当为0的时候，视为垃圾。

问题：循环引用 A引用了B B也引用了A

标记清除：划分了一个范围。比如：在树上、在内存中。当一个变量或者地址，被开辟时，则视为“进场”。当一个地址下树、不再被变量引用。则视为“出场”。

Promise：ES6中新增的一种处理异步代码书写风格的方式。

思想：定义一个Promise实例，监听一个异步请求。当异步请求成功时，将Promise实例的状态变为resolved。失败时，就将状态变为rejected。并同时触发对应的处理函数。

将代码从原来的异步请求的回调函数中，移动到了then方法中。

demo:

var p = new Promise(function(resolve, reject) {

$.get("/xxx", function() {

// 原来的处理代码是直接写在这里的。现在这里不再书写处理代码。而是仅仅判定成功失败。并调用resolve或者reject

})

})

p.then(function() {

真正的逻辑是放在这里

如果成功就执行这个函数

}, function() {

真正的逻辑是放在这里

如果失败就执行这个函数

})

Promise执行时，会返回一个Promise实例。并同时执行接收的函数参数。

返回值：Promise实例

then方法：

使用方式1：只监听一次。接收两个参数 分别表示成功时的回调和失败时的回调

使用方式2：连续调用

如果在连续调用的过程中，只有单个Promise实例。则每一个then都处理该Promise实例。并且只有第一个then会根据Promise实例的状态去决定调用成功或者失败处理函数。第二个、第三个、第四个……都只会执行then的第一个参数。

传参规则：第一个处理函数接收的参数是resolve或者reject执行时传递的值。后面的每一个处理函数接收的都是前一个处理函数的返回值。

如果在连续调用的过程中，前一个处理函数返回了新的Promise实例。则此时后面的then就不再监听第一个Promise实例。而是改为监听该新的Promise实例。又会根据新Promise实例的状态而决定调用成功或者失败处理函数。

catch方法：catch方法可以统一处理所有的Promise实例状态变为rejected时、或者Promise代码中同步代码的异常信息。

finally方法：finally方法是不论Promise成功还是失败。最终都会执行的方法。

如果一个Promise实例出错，但是没有写catch方法或者处理错误的方法。会向浏览器中抛出异常。

如果一个Promise实例状态已经发生变化，此时再给它绑定then方法还是可以触发。

状态变化：总共三个状态 pending、resolved/fulfilled、 rejected

状态只可能

从pending => resolved/fulfilled

或者

从pending => rejected

一旦状态确定，再也不会发生变化。

# 一、Promise的静态方法

静态方法：由类本身去调用的方法

普通方法：在类的原型上的方法，由实例调用

## 1.1 Promise.all

该方法用于同时监听多个Promise实例。返回一个新的Promise实例。当被监听的每一个Promise实例都状态变为resolved时，会将返回值的新Promise实例的状态变为resolved。当任何一个被监听的Promise实例状态变为rejected时，就会执行返回值的失败处理函数。

场景：

有两个请求，A、B。其中A和B之间没有任何关系。但是又处于同一套代码中。在没有Promsie.all时，我们只能先发送A，等A回来再发送B。或者先发送B，等B回来，再发送A。假设A耗时100ms B耗时120ms

则总耗时，220ms

如果有一种方式，能够允许它俩同时发送，并且能够在监听到两个都成功之后执行代码。则耗时大大减少。

Promise.all就是这样的一种方式。它监听多个Promise实例。

Promise.all(array);

array：数组 该数组的成员都应当是Promise实例。但是如果不是Promise实例也是可以的，它会使用Promsie.resolve静态方法将它转为Promise实例。

demo:

|  |
| --- |
| let p = Promise.all([task1(), task2()]);  p.then(function(arr) {  console.log("成功", arr); // arr是一个数组 分别是task1() 和task2状态变为resolved时传递的数据 位置一一对应  })  .catch(function(data) {  console.log("失败", data); // data是一个信息 是状态变为rejected的那一个Promise在调用reject时，传递的参数  }); |

输出：当成功时

|  |
| --- |
|  |

输出：当第一个失败时

|  |
| --- |
|  |

输出：当第二个失败时

|  |
| --- |
|  |

输出：计时

|  |
| --- |
|  |

比第二个请求的时间多一点点。远小于2个请求累加的时间。

## 1.2 Promise.race

Promise.race也是用于监听多个Promise实例。

Promise.all 返回的实例只有在所有的被监听的Promise实例都resolved之后，才会resolved。

Promise.race返回的实例只监听第一个返回的Promise实例的状态变化。

被监听了3个Promise实例。

如果其中某任何一个第一个发生状态的变化，Promise.race的返回值也会相应的发生变化。

简而言之：只监听第一个返回的Promise实例。

Promise.race(arr);

arr: 数组 数组中的每一个成员都是Promise实例。如果不是，会使用Promise.resolve转为Promise实例。但是此时，一定是监听它。

返回值：新的Promise实例

demo:

|  |
| --- |
| let p = Promise.race([task1(), task2()]);  p.then(function(data) {  console.log(data);  })  .catch(function(data) {  console.log(data);  }); |

输出：如果成功并且第一个先返回

|  |
| --- |
|  |

输出：如果成功并且第二个先返回

|  |
| --- |
|  |

输出：如果失败并且第一个先返回

|  |
| --- |
|  |

输出：如果失败并且第二个先返回

|  |
| --- |
|  |

结论：Promsie.race返回的实例跟随race接收的参数数组中第一个发生状态变化的Promise实例。

## 1.3 Promise.resolve

Promise.resolve(arg) 是一个Promise的静态方法 用于返回一个状态为resolved/fulfilled的Promise实例

arg: 可以是三种可能性

非Promise值

Promise实例

thenable对象 所谓的thenable对象指的是具备then方法的对象

### 1.3.1 非Promise值

demo:

|  |
| --- |
| 1. let p = Promise.resolve(123); 2. console.log(p); 3. p.then(function(data) { 4. console.log(data); 5. }) 6. .catch(function(data) { 7. console.log(data); 8. }); |

输出结果：

|  |
| --- |
|  |

结论：

当使用Promise.resolve将一个非Promise转为Promise时，得到一个Promise实例。该实例的状态是resolved/fulfilled。如果通过then方法监听，则会触发resolved/fulfilled时的处理函数。并且有传参，参数就是非Promise。

### 1.3.2 Promise实例

Promise.resolve如果接收的是一个Promise实例，则返回一个新的Promise实例，该Promise实例的状态跟随参数实例。

demo:

|  |
| --- |
| 1. // 通过Promise.resolve将task1的返回值转为Promise实例 2. let p = Promise.resolve(task1()); 3. // 其实等于 4. // let p = task1(); 5. p.then(function(data) { 6. console.log(data); 7. }) 8. .catch(function(err) { 9. console.log(err); 10. }); |

### 1.3.3 thenable对象

所谓的thenable对象指的是普通对象具备then方法

demo:

|  |
| --- |
| let p = Promise.resolve({  then: function(resolve, reject) {  setTimeout(function() {  // resolve("1");  reject("2");  }, 3000)  }  }); // 红色的对象具备then方法，所以是一个thenable对象  // 等价于  let p = new Promise(function(resolve, reject) {  setTimeout(function() {  // resolve("1");  reject("2");  }, 3000)  }); // 蓝色的部分其实就是红色对象的then方法 |

## 1.4 Promise.reject

该方法也是一个静态方法。返回一个状态为rejected的Promise实例。

参数是一个说明，说明为什么返回这个rejected。也就是“被拒绝”、“失败”的原因

demo:

|  |
| --- |
| 1. // Promise.reject 用于返回一个状态为rejected的实例 2. let p = Promise.reject("看你不顺眼"); 3. console.log(p); 4. p.catch(function(err) { 5. console.log(err); 6. }) |

输出：

|  |
| --- |
|  |

# Generator

它是ES6中增加的另外一种解决异步编程的方案。

## 2.1 感知Generator函数

定义：

|  |
| --- |
| 1. // 定义Generator函数 2. function \* Gen() { 3. // 它可以将一个函数分成多段执行 4. console.log("第一部分"); 5. yield; 6. console.log("第二部分"); 7. yield; 8. console.log("第三部分"); 9. yield; 10. console.log("第四部分"); 11. } |

调用：

|  |
| --- |
| 1. // 执行Gen 2. let g = Gen(); 3. console.log(g); |

输出：

|  |
| --- |
|  |

添加按钮并绑定事件：

|  |
| --- |
| 1. <button id="exe">点我执行一部分</button> 2. <script> 3. // 获取元素 4. let exe = document.getElementById("exe"); 5. exe.onclick = function() { 6. g.next(); 7. } 8. </script> |

每当点击该按钮一次，就会执行一部分Generator函数中的代码

|  |
| --- |
|  |

注：每一次点击按钮时，距离上一次的点击是一个异步行为。

关键点：

行为异步 代码书写同步

## 2.2 Generator函数的定义

定义语法：

function \* () {

yield;

yield;

yield;

yiled;

……

}

yield是一个ES6中新增的关键字。只能存在于Generator函数中。

调用：

当调用一个Generator函数时，会返回一个迭代器对象。此时Generator函数中的代码不会执行。

每当该迭代器对象执行next方法一次，就会执行一段Generator函数中的代码。从上一次的执行终点，到下一个yield关键字停止。

yield：它是一个关键字。

每当Generator函数执行到yield关键字时，就会交出线程的控制权。

g.next(): 这是一个迭代器对象的方法，每当它执行，就会再将线程的控制权交还回yield位置。

使用思路：

在每一个yield之间，发送异步请求。

在异步请求结束之后，调用g.next();

# 下午复习

Promise.all: 该方法用于监听多个Promise实例。

接收一个数组作为参数，数组中每一个都应当是Promise的实例。如果不是，会通过Promise.resolve方法将它转为Promise实例。

返回一个Promise实例

返回值的状态变化：

如果数组中的每一个Promise实例都resolved，则返回值变为resolved

如果数组中有任何一个Promise实例变为rejected，则返回值变为rejected

Promise.race: 该方法用于监听多个Promise实例。

接收一个数组作为参数，数组中每一个都应当时Promise的实例。如果不是，会通过Promise.resolve方法将他转为Promise实例。

返回一个Promise实例

返回值的状态变化：

跟随数组中第一个发生状态变化的Promise实例。

Promise.resolve: 该方法用于将一个值转为Promise实例

参数可能有三种情况：

第一种：参数是非Promise值、非thenable对象

返回的是一个resolved的Promise实例。传递的数据就是参数。

第二种：参数是Promsie实例

返回的是一个Promise实例，该Promise实例跟随参数的Promise实例的状态而变化

第三种：参数是thenable对象

返回的是一个Promise实例，该Promise实例的状态，要看thenable的then方法中调用resolve还是reject

Promise.reject: 该方法用于返回一个rejected的Promise实例。

参数：失败的原因描述 推荐使用字符串

Generator：

是一个状态机制。用于解决异步编程的问题。

Generator允许一个函数分段执行。

通过yield关键字进行分段。

每当执行到yield的时候，就交出线程的控制权。

定义：

function \* fun() {

yield;

yield;

yiled;

yield;

……

}

调用：

let g = fun();

g:是一个迭代器对象。

每当调用一次next方法，就让Generator函数中的一段代码执行。

从上一次停止的位置，执行到下一个yield。

## 2.3 Generator的简单使用

使用Generator的简单版本：

|  |
| --- |
| // 通过定义Generator函数 来发送两次AJAX请求  function \* Gen() {  // 发送第一次ajax  $.ajax({  url: "/ajax1",  dataType: "json",  success: function(data) {  console.log("发送完毕ajax1");  g.next(); // 当第一次ajax请求结束之后，就发送第二次  }  });  yield;  $.ajax({  url: "/ajax2",  dataType: "json",  success: function(data) {  console.log("发送完毕ajax2");  }  });  }  // 调用  let g = Gen();  g.next(); // 发送了第一次ajax请求 |

## 2.4 传递参数

第一种：从Generator函数内部向外部返回数据

第二种：从Generator函数外部向内部传递数据

### 2.4.1 从Generator函数内部向外部返回数据

内部的传递方式：yield关键字后面可以跟内容。这些内容会被返回出去。

外部的接收方式：g.next() 会返回一个对象，对象中有value有done 其中value就是yield给返回出来的内容。

demo:

|  |
| --- |
| // 定义Generator函数  function \* Gen() {  yield "你好";  yield "今天要打雷";  yield "今天要下雪";  yield "2012";  }  // 调用  let g = Gen();  let result = g.next();  console.log(result); |

输出：

|  |
| --- |
|  |

观察：每一次调用next方法都得到了里面向外返回的数据。

### 2.4.2 从Generator函数外部向内部传递数据

外部的传递方式：g.next()执行的时候可以传递参数

内部的接收方式：let 变量 = yield;

demo：

|  |
| --- |
| 1. // 定义Generator函数 2. function \* Gen() { 3. // 定义变量 接收外部传递的数据 4. // 因为是g.next()调用的时候会执行下一段代码 代码会从yield关键字左侧开始执行 5. // 所以传递的数据会被 yield关键字左侧部分接收到 具体的接收方式就是定义 6. // let 变量 = yield; 7. let a = yield; 8. console.log(a); 9. let b = yield; 10. console.log(b); 11. let c = yield; 12. console.log(c); 13. } 14. // 执行 15. let g = Gen(); 16. g.next(1);// 此时执行的是从Generator函数最开始到第一个yield关键字 所以等于没有执行任何代码 17. g.next(2);// 此时执行的是从第一个yield左侧 到第二个yield右侧 此时我们传递值 会被第一个yield左侧的变量接收到 18. g.next(3);// 此时执行的是从第二个yield左侧 到第三个yield右侧 此时我们传递值 会被第二个yield左侧的变量接收到 19. g.next(4);// 此时执行的是从第三个yield左侧 到第四个yield右侧 此时我们传递值 会被第三个yield左侧的变量接收到 20. // 执行完毕 |

执行结果：

|  |
| --- |
|  |

## 2.5 Co函数的使用

通过Generator函数，我们确实可以将不同的异步代码按照同步的方式去书写。

但是需要自己去书写逻辑交互。

demo: 糟糕的写法

|  |
| --- |
| 1. function task1() { 2. $.ajax({ 3. url: "/ajax1", 4. dataType: "json", 5. success: function(data) { 6. console.log(data); 7. g.next(data); 8. } 9. }) 10. } 11. // 定义第二个任务 12. function task2(data) { 13. $.ajax({ 14. url: "/ajax2", 15. data, 16. dataType: "json", 17. success: function(data) { 18. console.log(data); 19. } 20. }) 21. } 22. // 定义Generator函数 23. let gen = function \* () { 24. let a = yield task1(); 25. console.log(a); 26. yield task2(a); 27. } 28. let g = gen(); 29. g.next(); |

注意观察：g这个变量在全局中定义的没错，确实可以在task1中通过作用域访问到。但是关键在于：task1是一个函数，它跟gen这个Generator同级。并没有在内部定义g，也没有通过参数传递g。而是直接访问到g变量。这种写法很糟糕。

如果将这个函数移动到其它地方，不能用。因为未必有这个g变量。

于是我们找了一个Co函数，它可以帮助我们解除这个耦合现象

demo: co.js代码

|  |
| --- |
| function co(gen) {  // 返回值是一个Promise  return new Promise(function(resolve, reject) {  // 在promise中执行Generator函数  const g = gen();  // 定义一个函数  function next(nextFun) {  // 预先定义变量  let f = null;  // 尝试执行  try {  f = nextFun();  } catch(e) {  // 如果有错误，则被捕获到的错误信息被传递到rejected回调函数中  return reject(e);  }  // 判定是否执行结束  if (f.done) {  // 如果已经结束，此时才会执行返回的Promise实例的成功时的回调函数  return resolve(f.value);  }  // 如果没有结束，就将这一次的返回结果强行制作成一个Promise实例  Promise.resolve(f.value).then(function(result) {  // 如果f.value是一个值 则result就是该值  // 如果f.value是一个Promise实例 则result是它成功时的传递的参数  // 如果f.value是一个具备then方法的对象 则result是这个对象的then方法执行完毕之后的成功时的返回值  // 不论如何，result一定是成功时才会有的  // 也即是说，这个函数一定是在成功时才会执行的  // 如果成功了，就调用next继续下一次的执行  next(function() {  // 继续下一次的执行  return g.next(result);  });  }, function(result) {  // 如果f.value是一个值，则此函数不会执行  // 如果f.value是一个Promise实例 则result是它失败时的传递的值  // 如果f.value是一个具备then方法的对象 则result是这个then方法执行失败时的传递的信息  // 不论如何，result一定是失败时才会有的  // 也即是说，这个函数一定是在失败的时候才会执行的  // 如果失败了，就直接终止函数的执行  next(function(result) {  return g.throw(result);  });  })  }  // 开启第一次的next执行  next(function() {  return g.next(undefined);  });  })  } |

通过co函数来使用： 条件，yield之后跟的一定要是Promise实例。

|  |
| --- |
| // 定义任务  function taskOne() {  return new Promise(function(resolve, reject) {  $.ajax({  url: "/ajax1",  dataType: "json",  success: function(data) {  resolve(123);  }  });  });  }  function taskTwo() {  return new Promise(function(resolve, reject) {  $.ajax({  url: "/ajax2",  dataType: "json",  success: function(data) {  resolve(data);  }  });  });  }  // 定义Generator函数  function \* Gen() {  let data = yield taskOne();  console.log(data);  let data1 = yield taskTwo();  console.log(data1);  }  co(Gen); |

再看这种写法：

每一个任务中，都与其它任务不干涉。

## 2.6 g.throw

该方法用于向Generator函数抛出一个错误信息。

demo:

|  |
| --- |
| 1. // g.throw 用于向Generator中抛出一个错误 2. function \* Gen() { 3. try { 4. console.log(1); 5. // 定义yield 6. yield 1; 7. console.log(2); 8. yield 2; 9. console.log(3); 10. yield 3; 11. console.log(4); 12. yield 4; 13. console.log(5); 14. yield 5; 15. console.log(6); 16. yield 6; 17. console.log(7); 18. yield 7; 19. console.log(8); 20. yield 8; 21. } catch(e) { 22. console.log(e) 23. } 24. } 25. let g = Gen(); 26. g.next(); 27. g.next(); 28. g.next(); 29. g.throw("错误信息"); 30. g.next(); 31. g.next(); 32. g.next(); 33. g.next(); 34. g.next(); |

## 2.7 g.return

该方法用于主动将迭代器对象的迭代行为终止。

demo:

|  |
| --- |
| 1. function \* Gen() { 2. try { 3. console.log(1); 4. // 定义yield 5. yield 1; 6. console.log(2); 7. yield 2; 8. console.log(3); 9. yield 3; 10. console.log(4); 11. yield 4; 12. console.log(5); 13. yield 5; 14. console.log(6); 15. yield 6; 16. console.log(7); 17. yield 7; 18. console.log(8); 19. yield 8; 20. } catch(e) { 21. console.log(e) 22. } 23. } 24. let g = Gen(); 25. var result = g.next(); 26. console.log(result); 27. var result = g.next(); 28. console.log(result); 29. var result = g.next(); 30. console.log(result); 31. var result = g.return(123); 32. console.log(result); 33. console.log("哈哈哈哈"); 34. var result = g.next(); 35. console.log(result); 36. var result = g.next(); 37. console.log(result); 38. var result = g.next(); 39. console.log(result); 40. var result = g.next(); 41. console.log(result); 42. var result = g.next(); 43. console.log(result); |

红色代码部分，是主动将迭代器的迭代行为终止。将done属性变为true。  
后面代码依旧会执行，只是无法迭代到内容。

因为不再迭代，所以，next也就无法交还控制权给Generator函数内部了。

输出：

|  |
| --- |
|  |

## 2.8 return

普通函数中，可以通过return关键字返回一个数值给调用处。

Generator函数中，使用yield来返回数据给外部。return依旧可用。作用也是终止函数的执行。yield不会终止。yield是中止。

|  |
| --- |
| 1. // g.throw 用于向Generator中抛出一个错误 2. function \* Gen() { 3. console.log(1); 4. // 定义yield 5. yield 1; 6. console.log(2); 7. yield 2; 8. console.log(3); 9. yield 3; 10. console.log(4); 11. yield 4; 12. console.log(5); 13. console.log("终止"); 14. return "终止"; 15. yield 5; 16. console.log(6); 17. yield 6; 18. console.log(7); 19. yield 7; 20. console.log(8); 21. yield 8; 22. } 23. let g = Gen(); 24. console.log(g.next()); 25. console.log(g.next()); 26. console.log(g.next()); 27. console.log(g.next()); 28. console.log(g.next()); 29. console.log(g.next()); 30. console.log(g.next()); 31. console.log(g.next()); 32. console.log(g.next()); 33. console.log(g.next()); |

红色代码是在函数内部，将程序的执行终止。g.return是在函数外部将程序的执行终止。

## 2.9 for……of与g

g是一个迭代器对象，就可以被for……of去迭代。但是它只能迭代到yield的内容，而无法得到return的内容。

demo:

|  |
| --- |
| 1. function \* Gen() { 2. // 定义yield 3. yield 1; 4. yield 2; 5. yield 3; 6. yield 4; 7. yield 5; 8. yield 6; 9. yield 7; 10. yield 8; 11. return "终止"; 12. } 13. let g = Gen(); 14. for (var i of g) { 15. console.log(i); 16. } 17. let result = g.next(); 18. console.log(result); |

输出结果：最后的g.next() 执行的结果中，done已经变为true。说明真的是迭代完毕。而不是只迭代到最后一个yield。

|  |
| --- |
|  |

# async和await

## 3.1 简介

async是一个ECMAscript2016 也就是ES7的关键字。

它用于定义一个异步函数。表示“这个函数中有异步代码”。

await是一个ECMAScript2016 也就是ES7的关键字。

它用于放在函数前面，表示“等一等，等这个函数执行完毕再往下执行”

## 3.2 演示

|  |
| --- |
| 1. // async定义一个异步函数 它其实是Generator的语法糖 2. // await定义一个等待函数 它其实是yield的语法糖   // 语法糖：指的是对高级复杂特性的一种简化使用方式  async function taskOne() {  console.log(1);  await new Promise(function(resolve, reject) {  setTimeout(function() {  console.log("等待了3秒");  resolve();  }, 3000);  });  console.log(2);  await new Promise(function(resolve, reject) {  setTimeout(function() {  console.log("等待了3秒");  resolve();  }, 3000);  })  console.log(3);  }  // 执行这个异步函数  taskOne(); |

## 3.3 async

async定义一个异步函数 它其实是Generator的语法糖

定义语法： async function fun() {}

当定义了一个async函数之后，此时，函数的性质就变了。

普通函数的返回值，是由函数内的return关键字决定。

async函数的返回值，总是返回Promise实例。

demo:

|  |
| --- |
| // 定义普通函数  function fun() {  }  // 调用普通函数  let f = fun();  console.log(f); // undefined  // 定义异步函数  async function func() {    }  let f1 = func();  console.log(f1); // Promise的实例 |

输出：

|  |
| --- |
|  |

观察该Promsie，它的状态是resolved，它的值是undefined

**它的状态其实是由异步函数内的执行情况来决定。**

**如果一切正常，该函数内的所有代码能够顺利执行完毕。则状态变为resolved**

**如果有异常，则状态变为rejected**

**它的结果是由异步函数的返回值决定**

demo:演示出错时的情况

|  |
| --- |
| 1. async function func() { 2. console.log(1); 3. throw new Error("你好"); // 人为抛出错误 4. console.log(2); 5. } 6. let f1 = func(); 7. console.log(f1); // Promise的实例 8. f1.then(function(data) { 9. console.log("成功", data); 10. }) 11. .catch(function(err) { 12. console.log("出错", err); 13. }); |

Promise实例的状态：rejected

|  |
| --- |
|  |

如果不给Promise实例绑定错误处理代码：

|  |
| --- |
|  |

如果给绑定了错误处理代码：

|  |
| --- |
|  |

会捕获。

demo: 演示结果的变化

|  |
| --- |
| 1. // 定义异步函数 2. async function func() { 3. return 1; 4. } 5. let f1 = func(); 6. console.log(f1); // Promise的实例 7. f1.then(function(data) { 8. console.log("成功", data); 9. }) 10. .catch(function(err) { 11. console.log("出错", err); 12. }); |

输出：

|  |
| --- |
|  |

## 3.4 await

第一：await只能够出现在async函数中。就好像yield只能出现在Generator函数中。

第二：await表示“等待”。当执行到await的时候，会交出线程的控制权。

第三：await后面的内容

如果是Promise实例。则await左侧的变量接收到的值，就是Promise实例resolve()的传参。

如果不是Promise实例。则会使用Promise.resolve将它转为Promise实例。

demo1: 只能出现在async函数中

|  |
| --- |
| 1. // 定义async 2. async function fun() { 3. let a = await 123; 4. } |

此时，没有任何异常。但是如果将async去掉

|  |
| --- |
|  |

demo2: await表示“等待”。当执行到await的时候，会交出线程的控制权。

|  |
| --- |
| // 定义async函数  async function fun() {  console.log(123);  await 123;  console.log(321);  }  fun();  console.log("哈哈哈哈"); |

输出结果：

|  |
| --- |
|  |

demo3: await后面的内容

|  |
| --- |
| 1. // 定义async函数 2. async function fun() { 3. let a = await 123; 4. console.log(a); 5. } 6. fun(); |

输出：

|  |
| --- |
|  |

demo3.1：补充说明

await后面跟的是Promise实例，Promise实例如果成功，则await左边的变量可以得到数据。得到的值就是resolve时的数据。

|  |
| --- |
| // 验证  async function fun() {  let a = await new Promise(function(resolve, reject) {  resolve("hello world");  });  console.log(a);  }  fun(); |

注：resolve执行时传递的是一个字符串。所以await左边的变量a得到的就是一个字符串。

规则： resolve接收什么参数，await左边的变量就得到什么。

## 3.5 错误处理

如果代码中出现了错误，则会将异步函数返回值的Promise实例状态变为rejected并传递错误。

如果某一个await后面的Promise实例状态变为rejected。则也会将异步函数返回值的Promise实例状态变为rejected，并传递错误。

demo:

|  |
| --- |
| // 定义async函数  async function fun() {  throw new Error("123"); // 这一条会报错  let a = await new Promise(function(resolve, reject) {  setTimeout(function() {  reject(123); // 这一条也会报错  }, 3000);  });  }  // 如果某一个await后面的Promise实例状态变为了rejected  // 则直接中止函数的执行并将异步函数Promise实例的状态变为rejected  let p = fun();  p.catch(function(err) {  console.log(err);  }); |

输出：

|  |
| --- |
|  |

还有另外一种处理方式：

|  |
| --- |
| // 定义async函数  async function fun() {  // throw new Error("123");  try {  let a = await new Promise(function(resolve, reject) {  setTimeout(function() {  reject(123);  }, 3000);  });  }catch(err) {  console.log(err);  }  // 后续代码……  }  // 如果某一个await后面的Promise实例状态变为了rejected  // 则直接中止函数的执行并将异步函数Promise实例的状态变为rejected  let p = fun();  p.catch(function(err) {  console.log(err);  }); |

因为这里是在异步函数内部处理了错误，所以异步函数中的后续代码部分会继续执行。但如果没有try……catch。则会影响到异步函数的返回值的Promise状态变化。会导致后续代码部分不会继续执行

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |