

1. async函数

async函数是一个**语法糖**，使得异步操作更加简单。表现形式，在普通函数前面加上async关键字

async函数的返回值 是一个promise对象，async函数中return的值其实是 promise resolved时候的value，throw的值是promise rejected时候的reason，当没有显式的throw和return的时候，默认是return undefined

如以下代码示例一(return):

```
> async function test(){
  return 23
}

const p = test()
console.log(p)
p.then(value=>{
  console.log(value)
})
)
```

▼ Promise {<fulfilled>: 23} ⓘ VM1035:6

▶ __proto__: Promise

[[PromiseState]]: "fulfilled"

[[PromiseResult]]: 23

23 VM1035:8

上述代码等价于 function test(){ return Promise.resolve(1)}

代码示例二:

```
> async function test(){
  throw '错误'
}

let p = test()
console.log(p)
p.then(value=>{
  console.log(value)
},err =>{
  console.log(err)
})
)
```

▼ Promise {<rejected>: "错误"} ⓘ

▶ __proto__: Promise

[[PromiseState]]: "rejected"

[[PromiseResult]]: "错误"

错误

上述代码等价于 function test(){ return Promise.reject('错误')}

await 关键字

await是配合async函数一起使用的，只能出现在async函数内或者最外层，await在普通函数内部会报语法错误：如图所示



```
>
async function test(){
  await 1
}
< undefined

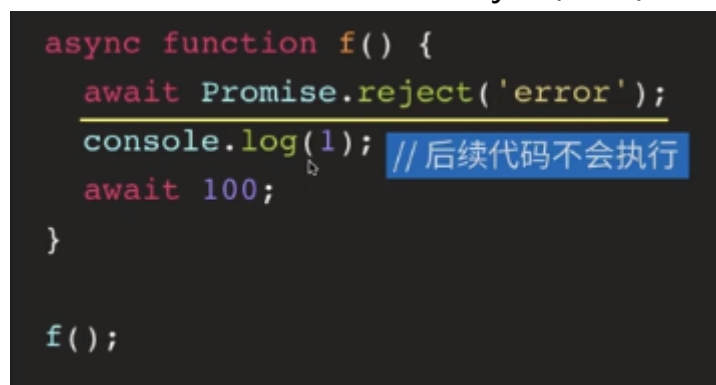
> test()
< Promise {<fulfilled>: undefined}

>
function test(){
  await 1
}
Uncaught SyntaxError: await is only valid in async function VM174:3
```

await的作用：等待一个promise对象的值，如果是promise的状态是resolved则等到resolved的值

await的promise状态为rejected，后续执行中断

例如下面示例：执行到Promise.reject(error)后面就不会执行了。



```
async function f() {
  await Promise.reject('error');
  console.log(1); // 后续代码不会执行
  await 100;
}

f();
```

如果希望后面的继续执行的话有两种方式：

第一种



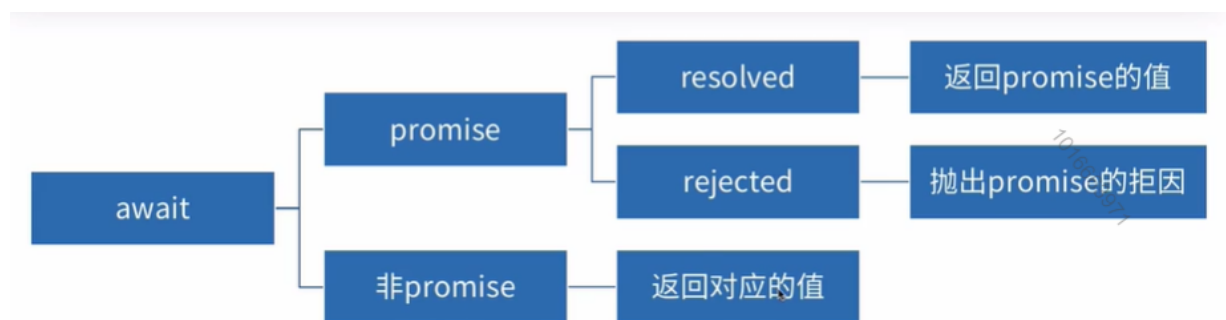
```
1 async function f() {
2   await Promise.reject("error").catch(err => {
3     // 处理异常
4   });
5   console.log(1);
6   await 100;
7 }
```

第二种

```
1 async function f() {  
2   try {  
3     await Promise.reject("error");  
4   } catch (e) {  
5     // 处理异常  
6   } finally {  
7  
8   }  
9   console.log(1);  
10  await 100;  
11 }
```

激活 Windows
转到“设置”以激活 Windows。

await 后面分两种 一种是promise对象 一种是是promise



1. `await` 一个promise的值的时候，promise状态为resolved，返回promise的value
2. `await` 一个promise的值的时候，promise状态为rejected，抛出promise的reason
3. `await` 一个非promise的值时，返回相对应的值，一般来说返回非promise无意义。
4. `await` 一个非promise的值时，`await`会阻塞后面的代码，先执行`async`外面的同步代码，再回到`async`内部，把这个非promise的东西，作为`await`表达式的结果。

5. await 一个promise的值时，await 也会暂停async后面的代码，先执行async外面的同步代码，等着 Promise 对象 fulfilled，然后把 resolve 的参数作为 await 表达式的运算结果

看示例代码一：

```
async function async1() {  
  console.log("async1 start");  
  await async2();  
  console.log("async1 end");  
}  
  
async function async2() {  
  return Promise.resolve().then(_ => {  
    console.log("async2 promise");  
  });  
}  
  
async1();
```

代码解析:

1. 执行大script宏任务，async1()调用，执行打印出 async1 start
2. 执行顺序是从右向做，执行async()函数的调用，async2返回一个promise对象，把微任务1丢到微任务任务队列，遇到await，中断await下面的代码的执行。先执行async外面的同步代码。执行完毕后，清空微任务队列，打印出async2 promise。
3. 继续回到await，再打印出async1 end
4. 事件循环结束

我们再回顾EventLoop机制的与async和await相关的那部分代码示例

最新的chrome改版后，老版本的有问题

rhinel commented on 17 Nov 2018

Owner

Author



Chrome 提交了优化ECMAScript编辑性更改:

- 1、await 后面不一定会创建新的微任务，取决于await 后面是立即返回还是promise。
- 2、await 执行之后不会强制创建新的微任务，而是继续执行。

以上两种修改会导致同步调用两个async函数时，执行权交换顺序会发生变化。
但是会提高性能。

来自 -- [Faster async functions and promises](#)

示例代码二:

```
1  async function async1() {
2    console.log( 'async1 start' )
3    await async2()
4    console.log( 'async1 end' )
5  }
6  async function async2() {
7    console.log( 'async2' )
8    return Promise.resolve().then(()=>{
9      console.log('测试')
10    })
11  }
12
13 console.log( 'script start' )
14 setTimeout( function () {
15   console.log( 'setTimeout' )
16 }, 0 )
17
18 async1();
19
20 new Promise( function ( resolve ) {
21   console.log( 'promise1' )
22   resolve();
23 } ).then( function () {
24   console.log( 'promise2' )
25 } )
```

代码执行分析:

script start async1 start async2 promise1 测试 promise2 async1 end

宏任务:setTimeout

微任务: '测试' 微任务 'promise2' 微任务 'async1 end'微任务

1. 进入大的script宏任务，执行打印出 script start

2. setTimeout推入宏任务队列
3. 执行async1, 打印出async1 start
4. await async2() 执行是从右到左, 执行async2(), 打印出async2, 同时返回一个Promise.resolve().then(()=>{ console.log('测试') }, resolve里没有什么, 所以什么都不打印, 遇到then, 推入微任务队列。
5. 执行await, 中断async函数, 执行async外的同步代码, 打印出promise1, 遇到then, 推入微任务队列。
6. 然后回到async函数, 继续执行。async2()返回的是一个新的Promise, 他的状态是根据新的Promise的去的, 他不是async函数默认包装出来的Promise, 也不是立即返回。所以这里创建了一个新的微任务
7. 执行栈清空, 然后检查所有的微任务, 分别得到执行结果 测试 promise2 async1 end
8. 第一轮宏任务结束, 执行第二轮宏任务, 打印出setTimeout

示例代码三:

```
1  async function async1() {
2    console.log( 'async1 start' )
3    await async2()
4    console.log( 'async1 end' )
5  }
6
7  async function async2() {
8    console.log( 'async2' )
9  }
10
11 console.log( 'script start' )
12 setTimeout( function () {
13   console.log( 'setTimeout' )
14 }, 0 )
15
16 async1();
17
18 new Promise( function ( resolve ) {
19   console.log( 'promise1' )
20   resolve();
21 } ).then( function () {
22   console.log( 'promise2' )
```

代码解析：

```
script start  async1 start  async2  promise1  async1 end  promise2
```

宏任务:setTimeout

微任务： '测试' 微任务 'promise2' 微任务 'async1 end'微任务

9. 进入大的script宏任务，执行打印出 script start
10. setTimeout推入宏任务队列
11. 执行async1，打印出async1 start
12. await async2() 执行是从右到左，执行async2()，打印出async2，同时返回一个Promise.resolve().then(()=>{ console.log('测试')})，resolve里没有什么，所以什么都不打印，遇到then，推入微任务队列。
13. 执行await，中断async函数，执行async外的同步代码，打印出promise1，遇到then，推入微任务队列。
14. 然后回到async函数，继续执行。async2()返回的是,async函数默认包装出来的Promise，立即返回。这里没有创建微任务，继续执行。打印出async1 end
15. 执行栈清空，然后检查所有的微任务，分别得到执行结果 测试 promise2
16. 第一轮宏任务结束，执行第二轮宏任务，打印出setTimeout

2. async函数实现的原理

Genertor+自动执行器

实现原理代码解析：（Genertor+自动执行器）

1. 根据async函数 返回的是一个promise对象
2. 所有在spawn函数的返回 new Promise()
3. spwan的参数是一个generator函数
4. spwan函数体内，gen是调用generator生成对象 step函数则是自动执行迭代器。
5. 直到迭代器执行完毕，resolved结果对象的value

3. async函数的应用

按顺序打印出多个文件的内容

我们前面有用到 回调方式 genertaor函数+Thunk函数方式 前面两种方式已经实现过了

我们再看一下promise方式

```
1 function readFilesByPromise() {
2     const fs = require("fs");
3     const files = [
4         "/Users/kitty/testgenerator/1.json",
5         "/Users/kitty/testgenerator/2.json",
6         "/Users/kitty/testgenerator/3.json"
7     ];
8     const readFile = function(src) {
9         return new Promise((resolve, reject) => {
10             fs.readFile(src, (err, data) => {
11                 if (err) reject(err);
12                 resolve(data);
13             });
14         });
15     };
16     readFile(files[0])
17         .then(function(data) {
18             console.log(data.toString());
19             return readFile(files[1]);
20         })
21         .then(function(data) {
22             console.log(data.toString());
23             return readFile(files[2]);
24         })
25         .then(function(data) {
26             console.log(data.toString());
27         });
28 }
29 // 调用
30 readFilesByPromise();
```

再看一下async加上await方式


```

1  async function readFilesByAsync() {
2      const fs = require("fs");
3      const files = [
4          "/Users/kitty/testgenerator/1.json",
5          "/Users/kitty/testgenerator/2.json",
6          "/Users/kitty/testgenerator/3.json"
7      ];
8      const readFile = function(src) {
9          return new Promise((resolve, reject) => {
10             fs.readFile(src, (err, data) => {
11                 if (err) reject(err);
12                 resolve(data);
13             });
14         });
15     };
16
17     const str0 = await readFile(files[0]);
18     console.log(str0.toString());
19     const str1 = await readFile(files[1]);
20     console.log(str1.toString());
21     const str2 = await readFile(files[2]);
22     console.log(str2.toString());
23 }
24 // 调用
25 readFilesByAsync();

```

激活 Windows
转到“设置”以激活 Wind

所以generator函数 + Thunk函数方式 promise方式 async + await
async+await方式更加清晰