09-第九章 ECMAScript6 async 函数

ECMAscript 6

> ES2017 标准引入了 `async 函数` ，使得异步操作变得更加方便。
> async 函数是什么？一句话，它就是 Generator 函数的语法糖。
> 一比较就会发现，async函数就是将 Generator 函数的星号（ * ）替换成 `async` ，将yield替换成 `await` ，仅此而已。

一、async函数特点

async函数对 Generator 函数的改进，体现在以下四点:

> （ 1 ） `内置执行器`
> Generator 函数的执行必须靠执行器(`next`)，所以才有了co模块，而async函数 `自带执行器` 。也就是说，async函数的执行，与普通函数一模一样，只要一行
>
> （ 2 ）更好的语义。
> async和 `await` ，比起星号和yield，语义更清楚了。async表示函数里有异步操作，await表示 `紧跟在后面的表达式需要等待结果` 。
> （ 3 ）更广的适用性
> co模块约定，yield命令后面只能是 Thunk 函数或 Promise 对象，而async函数的await命令后面，可以是 `Promise` 对象和 `原始类型的值` （数值、字符串和布尔值，但这时等同于同步操作）
>
> （ 4 ）返回值是 `Promise`
> async函数的返回值是 Promise 对象，这比 Generator 函数的返回值是 Iterator 对象方便多了。你可以用then方法指定下一步的操作

async函数完全可以看作多个异步操作，包装成的一个 `Promise 对象` ，而await命令就是内部then命令的语法糖

二、基本用法

(1) async关键字

```
async function fn(){};

//表示式
var  fn = async function(){};
//箭头
var fn = async()=>{};
//匿名

setTimeout( async function(){},1000)


// 对象的方法
let obj = { async foo() {} };
// Class 的方法
class Storage {
  constructor() {
    this.cachePromise = caches.open('avatars');
  }

  async getAvatar(name) {
    const cache = await this.cachePromise;
    return cache.match(`/avatars/${name}.jpg`);
  }
}

const storage = new Storage();
storage.getAvatar('jake').then(…);

```

函数前面的async关键字，表明该函数内部有异步操作。

---

(2) 调用async函数时，会立即返回一个Promise对象,状态为resolved

```
async function fn(){}
var r = fn();
 console.log( r );//Promise {[[PromiseStatus]]: "resolved", [[Pro
miseValue]]: undefined}
```

上面代码返回Promise对象，且状态为resolved

(3) return，会成为 PromiseValue值，状态为resolved

```
async function fn(){
    return 'hello'
}
var r = fn();
 console.log( r );//Promise {[[PromiseStatus]]: "resolved", [[Pro
miseValue]]: hello}
```

(4) 我们可以使用`then`方法指定async函数的回调函数

```
async function fn(){}
  fn().then(function(){
    console.log('ok')
  })
  //ok
```

return命令返回的值，会被then方法回调函数接收到

```
async function fn(){
  return 'hello'

 }
  fn().then(function(message){
    console.log(message)//hello
  })
```

async函数内部`抛出错误`，会导致返回的 Promise 对象变为`reject`状态。会触发reject回调函数，也可以被`catch`方法回调函数接收到。

```
async function fn(){
   throw new Error('出错了');

 }
fn().then(null,function(message){
  console.log(message)//Error: 出错了
})
```

```
 async function fn(){
   throw new Error('出错了');

 }
fn().then(function(message){
  console.log(message)//Error: 出错了
}).catch(function(e){
    console.log(e);
})
```

三、await 命令

从字面意思上看await就是等待，await 等待的是一个表达式,await命令后面是一个 `Promise 对象`。如果不是，async 会把这个直接量通过 `Promise.resolve()` 封装成 `Promise 对象`

很多人以为await会 一直等待之后的表达式执行完之后才会继续执行后面的代码 ，实际上await是一个让出线程的标志。await后面的函数会先执行一遍，然后就会跳出整个async函数来执行后面js栈（后面会详述）的代码。等本轮事件循环执行完了之后又会跳回到async函数中等待await
后面表达式的返回值，如果返回值为非promise则继续执行async函数后面的代码，否则将返回的promise放入promise队列（Promise的Job Queue）

```
async function fn(){
    console.log('await1');
     await '123';
     console.log('await2');
   }
   var t = fn();
    t.then(function(messa){
    console.log(messa);//123

   });
   console.log( t )
   //await1
   //{[[PromiseStatus]]: "pending", [[PromiseValue]]: "123"}
   //await2
```

```
async function fn(){
     return await '123';
   }
   var t = fn();
    t.then(function(messa){
    console.log(messa);//123
    console.log( t )//{[[PromiseStatus]]: "resolved", [[PromiseVa
lue]]: "123"}

   });
   console.log( t )//{[[PromiseStatus]]: "pending", [[PromiseValu
e]]: "123"}
```

上面代码中，await命令的参数是数值123，它被转成 Promise 对象，并立即resolve

---

await命令只能用在async函数之中，如果用在普通函数，就会报错

```
function fn(){
    await 'hello';//SyntaxError: Unexpected string
  }
```

四、Promise 对象的状态变化

`async函数`返回的 `Promise 对象`，必须等到内部所有`await`命令后面的 Promise 对象执行完，才会发生状态改变，除非遇到return语句或者抛出错误。也就是说，只有async函数内部的异步操作执行完，才会执行then方法指定的回调函数

```
async function getTitle(url) {
  let response = await fetch(url);
  let html = await response.text();
  return html.match(/<title>([\s\S]+)<\/title>/i)[1];
}
getTitle('https://tc39.github.io/ecma262/').then(console.log)
// "ECMAScript 2017 Language Specification"
```

（1）只要一个await语句后面的 Promise 变为`reject`，那么整个async函数都会`中断执行`

```
async function fn(){
      console.log('t1');
      await Promise.reject('hello 1');//Uncaught (in promise) hel
lo 1
      console.log('t2');
      await Promise.resolve('hello 2');
      console.log('t3');
   }
   fn();
```

（2）有时，我们希望即使前一个异步操作失败，也不要中断后面的异步操作。这时可以将第一个await放在`try...catch`结构里面，这样不管这个异步操作是否成功，第二个await都会执行。

```
async function fn(){
      try{
        await Promise.reject('hello 1');
      }catch(e){
      }
      await Promise.resolve('hello 2');
   }
   fn();
```

另一种方法是await后面的 Promise 对象再跟一个catch方法，处理前面可能出现的错误。

```
async function fn(){

        await Promise.reject('hello 1').catch(function(){});
        await Promise.resolve('hello 2');
    }
    fn();
```

（3）继发关系异步操作

```
let foo = await getFoo();
let bar = await getBar();
```

（4）同时触发的多个异步

如果getFoo和getBar是两个独立的异步操作（即互不依赖），被写成继发关系。这样比较耗时，因为只有getFoo完成以后，才会执行getBar，完全可以让它们同时触发。

```

async function fn(){
    let [foo,bar] = await Promise.all([Promise.resolve('hello'),Promise.resolve('ok')]);
    console.log(foo);//hello
    console.log(bar);//ok

 }
 fn();

```