# es8新特性

# 1.async和await

### 他们其实是Generator的语法糖，async函数返回一个Promise对象

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

### 需要注意，其实async不单独使用，他需要和await配合使用

|  |
| --- |
|  |

#### 模拟获取异步请求数据

|  |
| --- |
|  |

#### 由于使用async await这种写法没有catch分支，我们如果想捕获错误，需要把await代码写到try..catch语句块里面

|  |
| --- |
|  |

### 课堂案例，

#### 新建一个案例文件夹，在里面新建一个async-await4.html，把前面封装ajax的代码粘贴进来，然后新建一个1.json文件

|  |
| --- |
|  |

#### 1.json

|  |
| --- |
| {     "data":"123"  } |

#### async-await4.html

|  |
| --- |
| <body>      <script>           function myAjax(url) {              return new Promise(function(resolve,reject) {                  let xhr = new XMLHttpRequest()                  xhr.open("get",url,true)                  xhr.send()                  xhr.onreadystatechange = function () {                      if(xhr.readyState === 4){                          if(xhr.status>=200 && xhr.status<=300){                          resolve(JSON.parse(xhr.responseText))                          } else {                              reject(xhr.responseText)                          }                      }                    }           })           }           async function getAjaxData(){            let res =await myAjax("./1.json")            console.log(res);         }         getAjaxData()      </script>  </body> |
|  |

#### 还可以链式调用，我们新建一个2.json

|  |
| --- |
|  |

#### async-await4.html

|  |
| --- |
| <script>           function myAjax(url) {              return new Promise(function(resolve,reject) {                  let xhr = new XMLHttpRequest()                  xhr.open("get",url,true)                  xhr.send()                  xhr.onreadystatechange = function () {                      if(xhr.readyState === 4){                          if(xhr.status>=200 && xhr.status<=300){                          resolve(JSON.parse(xhr.responseText))                          } else {                              reject(xhr.responseText)                          }                      }                    }           })           }           async function getAjaxData(){            let res =await myAjax("./1.json")            console.log(res);            let res2 =await myAjax("./2.json")            console.log(res2);         }         getAjaxData()      </script> |
|  |

#### 也可以在await后面接Promise.all方法

|  |
| --- |
| <script>           function myAjax(url) {              return new Promise(function(resolve,reject) {                  let xhr = new XMLHttpRequest()                  xhr.open("get",url,true)                  xhr.send()                  xhr.onreadystatechange = function () {                      if(xhr.readyState === 4){                          if(xhr.status>=200 && xhr.status<=300){                          resolve(JSON.parse(xhr.responseText))                          } else {                              reject(xhr.responseText)                          }                      }                    }           })           }           async function getAjaxData(){          //   let res =await myAjax("./1.json")          //   console.log(res);          //   let res2 =await myAjax("./2.json")          //   console.log(res2);          //Promise.all          let res = await Promise.all([myAjax("./1.json"),myAjax("./2.json")])          console.log(res);         }         getAjaxData()      </script> |
|  |

## async await参考文档1.

## ES8 async/await: 优雅的异步编程解决方案

我相信, 在我们平时的开发过程当中, 大家或多或少的都会遇到要处理异步逻辑的情况, 最常见的情形比如是一个页面需要请求两个或者多个接口, 然后等这几个接口都返回之后再渲染页面, 这样的逻辑我们可以用回调函数的方式去完成, 以及ES6里的generator函数, 当然啦, 还有目前大家使用最多的, 也是ES6提供的另一个解决方案: promise对象, 回调函数的方式如果嵌套层数一多则会造成传说中的回调地狱问题, 代码冗余的同时不易维护, 而generator则被promise的光环盖过了, 目前在使用率上个人觉得是低于promise的, 不过蚂蚁金服的redux的框架[dvajs](https://link.juejin.cn/?target=https%3A%2F%2Fdvajs.com%2F)还在使用, 个人推测是历史遗留问题, 但仅仅是推测, 但话又说回来, dvajs使用generator来解决异步逻辑并没有什么不妥, 目前这套体系已经很成熟, 使用promise来重构duck不必

接下来就是promise, 说起promise想必大家都不陌生, 它是目前js异步逻辑最成熟的解决方案, 但如果聊到promis, 那么就一定离不开我们今天要聊的async/await, 因为没有它们, promise只是异步编程解决方案, 但有了它们才是优雅的异步编程解决方案

## promise

这不是本篇的重点, 但还是稍微聊一聊, 相信大多数的朋友对于这个语法已经很熟悉了, 如有不了解的童鞋可以查阅阮一峰老师的[ECMAScript 6 入门\_Promise 对象](https://link.juejin.cn/?target=https%3A%2F%2Fes6.ruanyifeng.com%2F%23docs%2Fpromise), 个人认为这是一篇不可多得的学习ES6相关语法的好文章

首先, Promise是一个构造函数, 需要我们使用new操作符调用, 从而生成一个promise对象, Promise接收一个函数作为参数, 同时这个函数还有两个参数, 这两个参数也是函数, 这两个函数依次是resolve和reject, 分别表示成功和失败:

scss

复制代码

const promise = new Promise(

(resolve, reject) => {

*//...*

if(/\* 异步操作成功 \*/) {

*//返回异步操作的结果*

resolve(value);

}else{

*//异步操作失败, 返回错误或者其他内容*

reject(error);

}

}

)

生成的promise对象上有一个then的方法, 这是成功的回调, 还有一个catch的方法, 这个方法则是失败的回调, 同时then方法会返回一个promise对象, 意味着我们可以使用jQuery那样的链式写法书写我们的代码:

ini

复制代码

promise

.then(

value => {

//...

}

)

.catch(

error => {

//...

}

)*;*

大多数情况我们会把promise对象当做一个函数的返回值来使用:

javascript

复制代码

**const** **handlePromise** = () => (

**new** **Promise**(

(resolve, reject) => {

*//...*

**if**(*/\* 异步操作成功 \*/*) {

*//返回异步操作的结果*

**resolve**(value);

}**else**{

*//异步操作失败, 返回错误或者其他内容*

**reject**(error);

}

}

)

);

**handlePromise**()

.**then**(

value => {

*//...*

}

)

.**catch**(

error => {

*//...*

}

);

## async/await

## 语法

接下来就是我们今天的主角: async/await, 这两个关键字是ES8的关键字, 配合ES6的promies就组成了目前js中最优雅的异步解决方案, 在此之前我们先来看看它们的语法是怎样的, 它们的语法很简单:

1. async将一般的函数变成了[异步函数](https://link.juejin.cn/?target=https%3A%2F%2Fdeveloper.mozilla.org%2Fzh-CN%2Fdocs%2FWeb%2FJavaScript%2FReference%2FStatements%2Fasync_function)
2. await只能在异步函数中使用, 同时它可以放在任何函数调用之前(这里主要是放在返回promise对象的函数调用之前)
3. 当await后面的函数有返回值, 那么我们将可以使用这个函数的返回值(如果那个函数返回的是promise对象, 则返回值是promise resolve返回的值, 也就是异步操作成功之后返回的值)

也就说它们是一起出现, 同时我们只能在async函数里面使用await关键字来接收函数的返回值

## async的用法

了解了语法, 那么我们来看看它们在实际业务中的写法, 首先看看async怎么用的, async是asynchronous的缩写, 异步的意思, 它会将普通函数改为异步函数, 写法有两种

### 函数声明/定义

就是使用function关键字声明一个函数, 将async写在函数声明之前, 那么这个普通的函数就变成了异步函数

csharp

复制代码

**async** function **foo**() {

*//...*

};

然后是模块化的写法:

javascript

复制代码

**export** **async** **function** **foo**() {

*//...*

};

javascript

复制代码

**export** **default** **async** **function** **foo**() {

*//...*

};

我个人的习惯是在导出class和组件的时候使用export default, 当只是一个util一个工具类函数的时候则使用export, 这个因人而异, 没有对错之分, 习惯使然, 只要注意export default和export的区别即可, 关于这两种导出方式的区别可以查看这篇文章: [ES6模块化import export的用法](https://juejin.cn/post/6965482913857028127)

### 函数表达式的写法

使用了ES6之后, 函数表达式就变成了我们写函数时最常用的方式了, 由于没有了function关键字, 此时异步函数的写法就稍有不同了:

javascript

复制代码

**const** **foo** = **async** () => {

*//...*

};

相应的模块化的语法则为:

javascript

复制代码

**export** **const** **foo** = **async** () => {

*//...*

};

## await的用法

聊完了async的用法, 接下来我们聊聊await的用法, await这个单词的意思就是等待的意思

### 写在普通函数调用之前

无论写在什么函数调用的前面, 都要留意它必须用在异步函数内:

ini

复制代码

const bar = () => {

console.log(123)*;*

}

const foo = async () => {

await bar()*;*

}*;*

foo()*;*

如果没有在async函数中写await, 而是在普通函数中写await:

ini

复制代码

const bar = () => {

console.log(123)*;*

}

const foo = () => {

await bar()*;*

}*;*

foo()*;*

此时将报错: await is only valid in async function

输出结果123, 倘若我们用一个变量去接收bar函数的返回值, 那结果是什么呢? 看代码我们会发现, bar函数并没有返回值, 所以结果是undefined:

ini

复制代码

const bar = () => {

console.log(123)*;*

}

const foo = async () => {

const res = await bar()*;*

console.log(res)*;*

}*;*

foo()*;*

此时先输出了123, 然后输出了undefined, 那如果我们让bar函数有一个返回值呢? 比如:

ini

复制代码

const bar = () => {

console.log(123)*;*

return 1*;*

}

const foo = async () => {

const res = await bar()*;*

console.log(res)*;*

}*;*

foo()*;*

此时则是先输出123, 然后再输出1, 我们发现我们接收到了bar函数的返回值, 然而平时我们在不写await的时候也能正常的获取某个函数的返回值:

ini

复制代码

const bar = () => {

console.log(123)*;*

return 1*;*

}

const foo = async () => {

const res = bar()*;*

console.log(res)*;*

}*;*

foo()*;*

这种情况下使用await似乎就没意义了, 这也解释了为何单独使用await会报错, 因为它只在返回promise的时候使用才有意义

### 写在promise函数调用之前

确切的说是写在返回promise对象的函数调用之前:

ini

复制代码

const bar = () => (

new Promise(

resolve => {

resolve(123)*;*

}

)

)

const foo = async () => {

const res = await bar()*;*

console.log(res)*;*

}*;*

foo()*;*

此时变量res的值是123, 也就是说我们的await会等待promise执行成功也就是执行resolve方法, 从而接收resolve方法的返回值, 如果不使用await, 那么我们就只是接收到一个promise对象而已

## 实现交通信号灯

回到最初的问题, async/await究竟优雅在什么地方呢? 那我们就要先看看单独使用promise不优雅在哪些地方, 上面的promise的代码其实还是使用了回调函数的写法, 也就是说如果有多个异步操作, 那么就会矫枉过正了, 会再回到我们的回调地狱中去, 得不偿失

比如我们要实现一个交通信号灯, 绿灯3秒, 黄灯1秒, 红灯2秒, 这样循环往复, 依次对比两种方法

## promise+then

scss

复制代码

const handleChangeColor = (color, time) => (

new Promise(resolve => {

console.log(color);

setTimeout(

() => {

resolve();

},

**time**

);

})

);

const handleTrafficLight = () => {

handleChangeColor('绿色', 3000).then(

() => {

handleChangeColor('黄色', 1000).then(

() => {

handleChangeColor('红色', 2000).then(

() => {

handleTrafficLight();

}

)

}

)

}

);

}

handleTrafficLight();

看着我们的main函数中的回调, 我整个人都开始不好了...

## async/await+promise

接下来我们把上面的写法改造一下:

javascript

复制代码

**const** **handleChangeColor** = (color, time) => (

**new** **Promise**(resolve => {

console.**log**(color);

setTimeout(

() => {

**resolve**();

},

time

);

})

);

**const** **handleTrafficLight** = **async** () => {

**await** **handleChangeColor**('绿色', 3000);

**await** **handleChangeColor**('黄色', 1000);

**await** **handleChangeColor**('红色', 2000);

**handleTrafficLight**();

}

**handleTrafficLight**();

效果一样, 但此时我们再看main函数中的写法, 一行接着一行, 用同步的写法实现了异步的逻辑, 看起来代码'清爽'了不少, 最重要的是逻辑更加清晰, '等待'一行执行结束再执行下一行, 再'等待', 再执行...代码量更少, 更易于维护, 这便是它的优雅之处了

# 2.es8的对象扩展语法

## 课堂笔记

### 对象的属性的get和set拦截的应用场景

|  |
| --- |
|  |

### 对象复杂属性的属性获取

|  |
| --- |
|  |

#### 注意，下面的复制属性不完全复制

|  |
| --- |
|  |

#### 这种方法只能复制属性，不能复制方法

### 解决办法就是使用Object. getOwnPropertyDescriptors()

|  |
| --- |
| let obj = {              name:"liu king",              age:30,              role:"street fighter",              email:"liuking123@gmail.com"          }            let obj1={}          Object.defineProperties(obj1,Object.getOwnPropertyDescriptors(obj))          console.log(obj1); |
|  |

#### 克隆对象推荐使用这种方法

## ES8对对象的扩展参考文档1

一、Object.values()：获取对象内值的数组

**const** obj **=** {

name**:**"lilei",

web**:**"www.baidu.com",

course**:**"math"

}

console.log(Object.values(obj)) *// ["lilei","www.baidu.com","math"]*

二、Object.entries()：返回二维数组，每个数组包含key和value

**const** obj **=** {

name**:**"lilei",

web**:**"www.baidu.com",

course**:**"math"

}

console.log(Object.entries(obj)) *// [["name","lilei"],["web","www.baidu.com"],["course","math"]]*

console.log(Object.entries(["a","b","c"])) *// [["0","a"],["1","b"],["2","c"]]*

三、循环遍历：与es6中的解构和字符串模板结合

**const** obj **=** {

name**:**"lilei",

web**:**"www.baidu.com",

course**:**"math"

}

**for**(**let** [key,val] **of** Object.entries(obj)){

console.log(`${key}:${val}`) *// name:lilei web:www.baidu.com course:math*

}

四、Object.keys（es8之前语法）

**const** obj **=** {

name**:**"lilei",

web**:**"www.baidu.com",

course**:**"math"

}

console.log(Object.keys(obj)) *// ["name","web","course"]*

**const** res **=** Object.keys(obj).map(key => obj[key])

console.log(res) *// ["lilei","www.baidu.com","math"]*

五、Object.getOwnPropertyDescriptors()：获取对象自身属性修饰符

**const** obj **=** {

name**:**"lilei",

course**:**"math"

}

**const** desc **=** Object.getOwnPropertyDescriptors(obj)

console.log(desc)

*// {*

*// name:{*

*// value:"lilei", // 默认值*

*// writable:true, // 属性是否可以改*

*// enumerable:true, // 是否可以通过for...in 循环遍历*

*// configurable:true // 是否可以通过delete删除掉*

*// },*

*// course:{*

*// value:"math",*

*// writable:true,*

*// enumerable:true,*

*// configurable:true*

*// }*

*// }*

六、与Reflect联合使用

**const** obj**=** {}

Reflect.defineProperty(obj,'name',{

value**:**"lilei",

writable**:false**,

configurable**:false**,

enumerable**:true**

})

Reflect.defineProperty(obj,'age',{

value**:**18,

writable**:false**,

configurable**:false**,

enumerable**:false**

})

console.log(obj) *// {name:"lilei"}*

obj.name**=** "zhangsan"

console.log(obj) *// {name:"lilei"}*

**delete** obj.name

console.log(obj) *// {name:"lilei"}*

**for**(**let** key **in** obj){

console.log(key) *// 只有name。没有age*

}

七、Object.getOwnPropertyDescriptor()

**const** obj **=** {

name**:**"lilei",

course**:**"math"

}

**const** desc **=** Object.getOwnPropertyDescriptor(obj,"name")

console.log(desc)

*// {*

*// value:"lilei", // 默认值*

*// writable:true, // 属性是否可以改*

*// enumerable:true, // 是否可以通过for...in 循环遍历*

*// configurable:true // 是否可以通过delete删除掉*

*// }*

八.利用Object.entries来创建map

|  |
| --- |
| let obj = {              name:"liu king",              age:30,              role:"street fighter",              email:"liuking123@gmail.com"          }  let objMap = new Map(Object.entries(obj))  console.log(objMap); |
|  |

# 3.es8字符串填充

|  |
| --- |
|  |

## 注意如果前面的数字小于或者等于字符串的长度，就不会填充

## 应用场景

|  |
| --- |
|  |
|  |

# 4.参数末尾加逗号

|  |
| --- |
|  |