**一、**

**ES6 Promise可以解决回调地狱、以及同步并发的异步问题。**

**（异步操作的异常捕获有其他方式解决。）**

但依旧会有明显的回调痕迹，之后学习ES6的generator 、ES7的

async await争取让异步代码看起来和同步一样。写起来更优雅一些。

**try catch 只能捕获同步代码出现的异常,异步的话无法捕获**

正常情况下 如果try中代码写错了

应该不会报错,会被catch捕获到

但是try中异步操作出现错误就会报错

1. **捕获异步的方法一**

在定时器的内部进行捕获

setTimeout(()=>{

try{

console.log(a);

}catch(e){

console.log(e);

}

})

1. **解决异步报错的方法二**

js代码一旦出现错误 就会执行该函数

**window.onerror** = function(){

console.log('hwt');

}

setTimeout(()=>{

console.log(a);

},1000);

console.log(a);

1. **Promise**
2. Promise是一个内置的构造函数

使用时候需要传递一个参数(执行器函数 === executor function)

注:执行器函数是同步执行,不是异步执行

1. promise是承诺的意思,承诺的事情一般都是异步的事情

promise是在执行器函数的里面进行异步的操作(异步的操作就是Promise所承诺的事);

异步操作操作完之后会有些相应的结果

根据结果去触发相应的回调

1. 执行器函数之中可以接受参数(resolve,reject);

分别用来触发oP对象注册的成功和失败的回调函数

1. 通过oP.then()方法进行回调函数的注册

类似于jQ中的then()方法,不过在Promise中,oP.then()之中只可以传两个参数

Promise只关心承诺的成功或失败,不会关心具体的等待过程

1. 执行器函数是同步执行,不是异步执行

但是执行函数中的resolve()和reject()触发的then()中的函数是异步执行的

then触发的函数不是同步执行,是异步执行的

1. then中用的函数放置在另一个任务队列中

该队列和setTimeout不是同一个队列

**then中函数异步执行会被当做微任务**,**会先被放到主线程的执行栈中去执行**

then中用的是另一个异步函数被promise()内置执行

**在异步操作中分两种任务 宏任务 微任务**

宏任务:setTimeout、ajax回调、触发事件函数 task queue

宏任务会被优先放置到任务队列1中 task queue1

微任务会被后被放置到任务队列2中 task queue2

**注:微任务有优先执行权,即使后被放置到任务队列中,但是微任务有优先执行权**

**微任务会先被拿出来放到执行栈中执行**

1. **在对then进行链式调用的时候**

如果上一个then不抛出错误,那下一个then执行成功的函数

**(默认触发then中的第一个函数)**

如果上一个then抛出错误,那么下一个then执行失败的函数

**(默认触发then中的第二个函数)**

注：只要上一个then中的注册函数抛出错误,那么下一个then就执行失败的函数

注:

1. **上一个then返回一个不是promise对象的值 会被当做下一个then中的参数**

**返回值作为下一个then注册函数的执行参数**

1. **如果上一个then的返回值是Promise对象**

返回的Promise对象就会触发下一个then中的注册函数

**下一个then就不会默认执行失败或者成功的函数**

**根据Promise对象触发的函数来决定执行哪个函数**

1. **Promise中的Api**
2. **Promise.catch()进行错误捕获**

oP.catch()可以进行错误的捕获

**除了catch, 还可以通过Promise()进行数据捕获**

**可以通过下一个then()中的失败函数进行错误的捕获**

通过链式调用then()只要后面then中有一个失败函数,就可以进行错误的捕获

链式调用的时候 如果写一个空then其实相当于不存在可以忽视

1. **在catch()后面.then()**

**如果上面catch()中不抛出错误**

**then()中的成功函数就会执行**

**如果前面抛出的错误被then()中失败函数捕获到**

**那么catch()就相当于一个空then() 可以忽视**

如果前面没有错误函数捕获

那么then可以正常捕获和并且返回值给下一个then的形参

1. **Promise.finally();**

无论是成功还是不成功,最后都会执行的东西

对oP中的链式调用进行终结的方法

finally()中没有参数

对链式调用进行收官处理

之后尽量不要再.then();

1. **Promise.all();**

**里面的参数必须是数组、数组中必须放置一堆Promise对象**

**Promise.all()的返回值也是一个Promise对象**

**当里面的Promise对象全部触发成功,就会执行成功的回调函数**

**有一个触发失败了,就会执行失败的回调函数**

成功的时候返回的是一个数组

而失败的时候则返回最先被reject失败状态的值

(谁先触发失败,就会返回谁的值)

1. **Promise.race()**

**Promise.race()的返回值也是一个Promise对象**

**只要有一个Promise对象触发成功 就会触发成功函数**

**只要有一个Promise对象触发失败 就会触发失败函数**

race()就是赛跑的意思,谁的状态先发生改变就输出谁的值

只要有一个Promise对象先触发成功或者失败

就会执行相应的函数

1. **Es6中新增的属性Symbol**

Es6变化

1. **Iterator:迭代器**

迭代模式：

提供一种方法可以顺序获得聚合对象中的各个元素，是一种最简单也最常见的设计模式。它可以让用户透过特定的接口巡访集合中的每一个元素而不用了解底层的实现。

例如：forEach()就是一个迭代器 内部迭代器

代器简介：

依照与迭代模式的思想而实现，分为内部迭代器和外部迭代器。

迭代 == 遍历

**内部迭代器：**本身是函数，该函数内部定义好迭代规则，完全接手整个迭代过程，

外部只需要一次初始调用。

Array.prototype.forEach 、jQuery.each 内部迭代器

**外部迭代器：**本身是函数，执行返回迭代对象，迭代下一个元素必须显示调用，调用复杂度增加，但灵活性增强。

function outerItreator(){} 外部迭代器

**迭代器目的：**

**从迭代模式思想中可以看出，就是要标准化迭代操作。**

**举个例子： [迭代器就是用来解决数据解构不同的数据遍历问题]**

服务端提供数组数据给前端，前端for循环遍历，但由于业务变化，使得数据结构发生变化，返回对象或者set map，导致前端遍历代码大量重写。

解决方案:

ES6引入Iterator，**部署在NodeList、arguments、Array, Set,**

**Map、字符串上等数据的Symbol.iterator属性。**

之前学习Set、Map方法的时候提到过迭代接口

Set、Map、for...of循环、传递参数必须带有迭代接口

使得这些数据是iterable可迭代的，能进行for of、... 、

Array.from等操作 （也可以转化字符串为数组

**迭代器引入的目的:**

**可以对不同的数据进行统一的遍历,避免代码的重构**

二、

**Symbol.iterator**

数据类型 String number undefined null boolean object

**新增的第七种数据结构:Symbol**

特点:

**唯一,可作为对象的属性,有静态属性Symbol.iterator**

1.

如果Symbol中传递的参数是一个对象

那么就会调用toString()方法,返回的值放在Symbol中

let os = Symbol({

name:'hwt',

toString:function(){

return 'duyi';

}

})

**返回 Symbol(‘duyi’)**

1. **Symbol具有唯一性 Symbol('a') != Symbol('a')**
2. **Symbol作为对象的属性**

let os = Symbol('a');

let name = 'name';

let obj = {

[os] :'hwt',

[name]:'ww',

}

取到属性值: obj[os]

4.Symbol上的静态属性 --》 Symbol.interator

arr Set Map " " arguments nodelist

每一个数据原型上都有Symbol(Symbol.iterator)属性

并且属性都 == iterator函数

console.log(Symbol.iterator,Symbol('Symbol.interator'));

三、

**给没有被部署iterator函数的数据添加迭代接口**

**实现iterator函数有自己的规则**

Iterator实现规则：

**1.有一个返回值:并且返回值是一个对象**

**2.对象中必须有一个next方法**

next、value、done

对象中部署Iterator：

let obj = {[Symbol.iterator]: function iteratorFunc() {}};

// 里面的参数必须满足Es6中的规范

**注:必须满足Es6中的规范**

**(1).value,done的属性名不能改变**

**(2).返回的对象中的方法名也必须是next,不能改变**

var obj = {

0:'hwt',

1:'age',

2: 175,

length:3,

[Symbol.iterator]:function(){

var curIndex = 0;

let next = ()=>{

return {

value:this[curIndex],

done:this.length == ++curIndex,

}

}

return {

next:next,

}

}

}

1. **Es6中的Generator生成器**

**Generator简介:**

生成器，本身是函数，执行后返回迭代对象，函数内部要配合yield使用

Generator函数会分段执行，遇到yield即暂停,

再次调用迭代对象中的next()方法就会继续向下执行

特点：

function和函数名之间需要带 \*

函数体内部yield表达式，产出不同的内部状态 (值)