# ES6

## 箭头函数

Es6标准新增了一种新的函数：Arrow Function（箭头函数）

箭头函数相当于匿名函数，并且简化了函数定义。箭头函数有两种格式，一种只包含一个表达式，可以省略{……}和return。还有一种可以包含多条语句，这时候不能省略{……}和return。

* 箭头函数this总是指向词法作用域，也就是外层调用者。

使用call、apply无法改变箭头函数的this。

* 箭头函数不能作为构造函数，即不可以使用new实例化。
* 箭头函数不绑定arguments。
* 箭头函数没有prototype属性。
* 箭头函数不能用作函数生成器，yield关键字不能在箭头函数中使用。

## 数组—扩展运算符（…）

将一个数组转化为用逗号分割的参数序列。

## 数组—flat

嵌套数组转一维数组 flat()

var arr = [1,2,[3,4,[5,6]]]

console.log(arr.flat(3))

输出： [ 1, 2, 3, 4, 5, 6 ]

## Proxy

## Reflect

## Babel、polyfill

Babel默认只转换新的JavaScript句法（syntax），二不转化新的API，比如Iterator、Generator、Set、Map、Proxy、Reflect、Symbol、Promise等全局对象，以及一些定义在全局对象上的方法（比如Object.assign）都不会转码。

## Promise

1. Promise是异步编程的一种解决方案；
2. Promise对象代表一个异步操作；
3. Promise对象有三种状态：pending（进行中）、fullfilled（已完成）、rejected（已失败）；
4. Promise构造函数接受一个函数作为参数，该函数称之为“构造器函数”
5. 构造器函数接受两个参数，分别是resolve和reject。它们是两个函数，有JavaScript引擎提供；
6. resolve函数的作用是，将promise对象的状态从“进行中”变成“已完成”，在异步操作成功时调用，并将异步操作结果作为参数传递出去；
7. reject的作用是，将Promise对象的状态从“进行中”变为“已失败”，在异步操作失败时调用，并将异步操作报出的错误，作为参数传递出去；

**promise.prototype.then**

1. Promise实例生成以后，可以用then方法分别指定resolved状态和rejected状态的回调函数；
2. then方法可以接受两个回调函数作为参数。第一个回调函数是promise对象的状态变成resolved时调用，第二个回调函数是promise对象的状态变为rejected时调用。这两个函数都是可选的，不一定要提供；
3. then方法返回的是一个新的Promise实例。因此可以采用链式写法，即then方法后面再调用另一个then方法。

<https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Reference/Global_Objects/Promise/all>

Promise.prototype.catch()

Promise.prototype.finaly()

Promise.all()

Promise.race()

Promise.allSettled()

Promise.any()

Promise.resolve()

Promise.reject()

## Promise的缺点

1. 无法取消Promise；
2. 如果不设置回调函数，Promise内部抛出的错误，不会反应到外部；
3. 当处于pending状态时，无法得知目前进展到哪一个阶段（刚刚开始还是即将完成）

## promise.then()方法里面的函数为什么是异步的？

假设、promise.then()内部的方法是异步执行的；那么then内部的异步函数肯定在promise内部的异步函数之后执行。

Promise.then通过将内部方法push到一个数组里面进行订阅

当promise里面的异步函数执行完成，会调用上面的数组执行回调函数，由于then里面的方法还没有执行，只是数组则为空。

## 手写promise

const PENDING = 'pending'

const RESOLVED = "resolved"

const REJECT = "reject"

//因为所有的promise都遵循这个规范，所以

const resolvePromise = (promise2, x, resolve, reject) => {

   //判断X的值

   if(promise2 === x){ //此时会出现死循环，promise2不能return promise2

       return reject(new TypeError('promise2不能返回promise2'))

   }

   //判断数据类型 typeof contructor instanceof toString

   if(typeof x === 'object' && typeof x !== null || typeof x === 'function'){

        let called = false;

        try {

            let then = x.then

            if(typeof then === 'function'){ //认为它是promise

                then.call(x, yes=>{

                    if(called){

                        return

                    }

                    called = true

                    // resolve(yes) //如果yes还是一个promise

                    resolvePromise(promise2, yes, resolve, reject)

                }, no =>{

                    if(called){

                        return

                    }

                    called = true

                    reject(no)

                })

            }else{

                resolve(x)

            }

        }catch (e){

            if(called){

                return

            }

            called = true

            reject(e)

        }

   }else{

       // x是一个普通值

       resolve(x)

   }

}

class Promise{

    constructor(executor){

        this.status = PENDING;

        this.value = undefined;

        this.error = undefined;

        this.onfulfilledcallback = []

        this.onrejectedcallback = []

        let resolve = (value) =>{

            if(this.status == PENDING){

                this.status = RESOLVED

                this.value = value

                this.onfulfilledcallback.forEach(fn=>{

                    fn()

                })

            }

        }

        let reject = (error) => {

            this.status = REJECT

            this.error = error

            this.onrejectedcallback.forEach(fn=>{

                fn()

            })

        }

       try {

            executor(resolve, reject);

       } catch (error) {

           reject(error)

       }

    }

    then(onfulfilled, onrejected){

       //同步的情况

       //同步时resolve在then前面执行

        if(this.status == RESOLVED){

            onfulfilled(this.value)

        }

        if(this.status == REJECT){

            onrejected(this.error)

        }

        //异步的情况

        //异步时then在resolve前面执行

        if(this.status == PENDING){

            this.onfulfilledcallback.push(()=>{

                onfulfilled(this.value)

            })

            this.onrejectedcallback.push(()=>{

                onrejected(this.error)

            })

        }

    }

    //链式

    // 1、判断成功和失败函数的返回结果

    // 2、判断是不是promise，就采用他的状态

    // 3、如果不是promise

    then2(onfulfilled, onrejected){

        //onfulfilled, onrejected是可选参数

       onfulfilled = typeof onfulfilled === 'function'? onfulfilled: data => {

           return data

       }

       onrejected = typeof onrejected === 'function'?onrejected : err => {

           throw err

       }

        //同步的情况

       //同步时resolve在then前面执行

       let promise2 = new Promise((resolve, reject)=>{

            if(this.status === RESOLVED){

                setTimeout(() => {

                    try {

                        let x = onfulfilled(this.value)

                        // x可能是普通值，也可能是promise

                        // 此处使用promise2，需要使用异步的方式

                        // 加定时器后，异步代码没办法被外部try-catch捕获

                        resolvePromise( promise2, x, resolve, reject)

                    } catch (error) {

                        reject(error)

                    }

                },  0)

            }

            if(this.status === REJECT){

                setTimeout(() => {

                   try {

                        let x = onrejected(this.error)

                   } catch (error) {

                       reject(error)

                   }

                }, 0);

            }

            //异步的情况

            //异步时then在resolve前面执行

            if(this.status === PENDING){

                this.onfulfilledcallback.push(()=>{

                    setTimeout(() => {

                        try {

                            let x = onfulfilled(this.value)

                            resolvePromise( promise2, x, resolve, reject)

                        } catch (error) {

                            reject(error)

                        }

                    }, 0);

                })

                this.onrejectedcallback.push(()=>{

                    setTimeout(() => {

                        try {

                            let x = onrejected(this.error)

                            resolvePromise( promise2, x, resolve, reject)

                        } catch (error) {

                            console.log("异步出错")

                            reject(error)

                        }

                    }, 0);

                })

            }

       })

        return promise2

    }

}

const isPromise = (value) => {

    if((typeof value === 'object' && value !== null) || typeof value === 'function' ){

        if(typeof value.then === 'function'){

            return true

        }

    }

    return false

}

Promise.prototype.finally = function(cb){

    return p.then(data => {

        Promise.resolve(cb()).then(()=>data)

    }, err => {

       Promise.resolve(cb()).then(()=> {

           throw err

       })

       throw err

    })

}

Promise.all = function(values){

    return new Promise((resolve, reject)=>{

        let arr = []

        let index = 0

        function processData(key, value){

            arr[key] = value

            if(++index === values.length){

                resolve(arr)

            }

        }

        for(let i=0; i<values.length;i++){

            let current = values[i]

            if(isPromise(current)){

                current.then((data)=>{

                    processData(i,data)

                }, reject);

            }else{

                processData(i,current)

            }

        }

    })

}

Promise.all([1,2]).then((data)=>{

    console.log(data)

})

let p = new Promise(function(resolved, reject){

    setTimeout(() => {

        resolved('成功')

    }, 1000);

})

p.then2((data) =>{

    console.log('1是一个字符串：', data)

    return new Promise((resolve, reject) =>{

        resolve(123)

    });

}).then2((data)=>{  //订阅，先订阅再发布

   console.log('2是一个promise',data)

   return ()=>{

       console.log("函数输出")

   }

}, (error) => {

    console.log('---', error)

}).then2(data => {

    console.log('3是一个函数', data)

}).then2(data => {

    console.log(4, data)

})

## 协程

协程是一种程序运行的方式

协程是一种比线程更加轻量级的存在，协程存在线程的环境中，一个线程可以存在多个协程，可以将协程理解为线程中的一个个任务。不像进程和线程，协程不受操作系统的管理，而是被具体的应用程序代码所控制。一个线程一次只能执行一个协程，比如当前执行A协程，另外还有一个B协程，如果想要执行B的任务，就必须在A的协程中将JS线程的控制权转交给B协程，那么现在B执行，A就相当于处于暂停的状态。A将执行权交给B，也就是A启动B，我们也称A是B的父协程

## Generator 生成器

[ˈdʒenəreɪtə(r)]

* function关键字后跟一个星号会定义一个生成器函数(generator function)；
* 执行Generator函数会返回一个遍历器对象；
* Generator函数是分段执行的，yield表达式是暂停执行的标记，next方法可以恢复执行；
* yield表达式后面的表达式，只有当调用next方法，才会执行；
* next方法

1、next方法的作用是分段执行generator函数；

2、next() 方法返回一个对象，这个对象包含两个属性：value和done；

（1）value属性是yield语句后面表达式的值，表示当前阶段的值；

（2）done属性为布尔值，表示generator函数是否执行完毕，即是否还有下一个阶段；

3、next方法可以传递参数，向generator函数体内输入数据；

* 错误处理机制

1. generator函数体外，使用指针对象的throw方法抛出错误，可以被函数体内的try-catch代码块捕获。
2. 出错的代码和处理错误的代码，实现了时间和空间上的分离。

* yield\*
* **与Iterator的关系**

[yi ter rei ter]

Symbol [ˈsɪmbl]

* **for-of循环遍历generator实例对象**

function \* gen(){

    yield 1;

    yield 2;

    yield 3;

}

var g = gen()

for(let i of g){

    console.log(i)

}

输出为： 1、2、3

* **generator.prototype.throw**

[ˈprəʊtətaɪp]

Generator函数返回的遍历器对象，都有一个throw方法，可以在函数体外抛出错误，然后在generator函数体内捕获

* **generator.prototype.return**
* **yield\* 表达式**

[jiːld]

如果在generator函数内部，调用另一个generator函数。需要在前者的函数体内部，手动完成变量。

function\* genA(){

    yield 1;

    yield 2;

}

function\* genB(){

    yield 'a';

    for(let i of genA()){

        console.log(i)

    }

    yield 'b'

}

for(let i of genB()){

    console.log(i)

}

输出结果：

a

1

2

b

如果有多个generator函数嵌套，写起来就非常麻烦。ES6提供了yield\*表达式，作为解决办法，用来在一个generator函数里面执行另一个generator函数。

function\* genA(){

    yield 1;

    yield 2;

}

function\* genB(){

    yield 'a';

    yield genA();

    yield 'b'

}

for(let i of genB()){

    console.log(i)

}

输出结果：

a

Object [Generator] {}

b

## async/await

**1．async与generator的关系**

async函数是generator函数的语法糖

async函数读generator函数的改进，体现再以下四点：

（1）内置执行器

Generator函数的执行必须靠执行器，而async函数自带执行器。

（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）更广的适应性

* yield命令后面只能是Thunk函数或Promise对象；
* await命令后面表达式的结果，就是await要等的东西

（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。你可以使用then方法指定下一步的操作。

**2．async关键字**

* async作为一个关键字放在函数前面，用于表示函数是一个异步函数；异步函数就意味着该函数的执行不会阻塞后面代码的执行。
* async函数返回的是一个promise对象。
* 如果async函数中有返回一个值，当调用该函数时，内部会调用Promise.resolve()方法把它转化成一个promise对象作为返回。如果函数内部抛出错误，就会调用Promise.reject()方法返回一个promise对象。

**3. await关键字**

* await关键字只能放在async函数里面。
* await后面跟着什么？await后面可以放任何表达式，不过我们更多的是放一个返回Promise对象的表达式。
* await是等待的意思，那么它等待什么呢？await等后面的promise对象执行完毕，然后拿到promise resolve的值并进行返回，返回值拿到之后，它继续向下执行。

1. **错误处理**

* 如果await后面的异步操作出错，那么等同于async函数返回的Promise对象被reject。
* 防止出错的方法，也是将其放在try…catch代码块之中

1. **async函数的实现原理**

async函数的实现原理，就是将generator函数和自动执行器，包装在一个函数里。

async function fn(args){

}

        function fn(args) {

            return spawn(function\* (){

            })

        }

        function spawn(genF) {

            return new Promise(function(resolve, reject){

                const gen = genF();

                function step(nextF){

                    let next;

                    try {

                        next = nextF();

                    } catch (e) {

                        return reject(e);

                    }

                    if(next.done){

                        return resolve(next.value)

                    }

                    Promise.resolve(next.value).then(function(v){

                        step( function(){ return gen.next(v); });

                    }, function(e){

                        step( function(){ return gen.throw(e); })

                    })

                }

                step(function(){ return gen.nextF(undefined)})

            })

        }

## Promise、generator、async/await比较

* Promise的写法比回调函数的写法大大改进，但是一眼看上去，代码上都是Promise的API，操作本身的语义反而不容易看出来；
* Generator函数的语义比Promise写法更清晰，但需要用户自己维护一个自动执行器；
* Async函数的实现最简洁，最符合语义，几乎没有语义不相关的代码，它将generator写法中的自动执行器，改在语言层面提供，不暴露给用户，因此代码量最少。

## Java的class和JavaScript的class有什么区别？

javascript的类是基于原型的继承机制；

java的类是基于类的继承机制。

## Iterator遍历器

遍历器（Iterator）是一种接口，为不同的数据结构提供统一的访问机制。

任何数据结构只要部署了Iterator接口，就可以完成变量操作

<https://es6.ruanyifeng.com/>

## for in 和 for of的区别

* for in 多用于对象的遍历，若是用来遍历数组，则拿到的是数组的索引。若是用来遍历对象，拿到的是属性名。

for of多用于遍历数组，拿到的是数组的每一项。若是用来遍历对象，则会报错（TypeError：obj is not iterable），因为对象不可迭代，没有迭代器。

## JS错误捕获

<https://segmentfault.com/a/1190000014672384>

前端错误分为JS运行时的错误、资源加载错误和接口错误三种。

1. 运行时的错误

Js运行时错误一般使用window.onerror捕获，但是有一种特殊情况就是promise被reject并且错误信息没有被处理的时候抛出的错误

* 1. 一般情况下的JS运行错误

使用window.onerror和window.addEventListener(‘error’)捕获。

* 1. Uncaught (in promise)

当promise被reject并且错误信息没有被处理的时候，会抛出一个unhandledrejection,并且这个错误不会被window.onerror以及window.addEventListener(‘error’)捕获，需要用专门的window.addEventLister(‘unhandledrejection’)捕获处理。

* 1. console.error

一些特殊情况下，还需要捕获处理console.error，捕获方式就是重写window.console.error

* 1. 特别说明跨域日志

当加载不用域的脚本中发生语法错误时，为避免信息泄露，语法错误的细节将不会报告，而代之简单的“script error”。在某些浏览器中，通过在<scrpit>使用crossorigin属性并要求服务器发送适当的CORS HTTP响应头

* 1. 特别说明sourseMap

在线上由于JS一般都是被压缩或者打包过，打包后的文件只有一行，因此报错会出现第一行第5000列出现JS错误，给排查带来困难。SourceMap存储打包前的JS文件和打包后的JS文件之间一个映射关系，可以根据打包后的位置快速解析出对应源文件的位置。

但是出于安全性考虑，线上设置sourceMap会存在不安全的问题，因为网站使用者可以轻易的看到网站的源码，此时可以设置.map文件只能通过公司内网访问减低隐患

SourceMap配置devtool:’inline-source-map’

如果使用了uglifyjs-webpack-plugin必须把sourceMap设置为true。

* 1. 其他
     1. Sentry 把所有的回调函数使用try-catch封装

<https://github.com/getsentry/raven-js/blob/master/src/raven.js>

* + 1. vue errorHandler

<https://vuejs.org/v2/api/#errorHandler>

其原理也是使用try-catch封装了nextTick、$emit、watch、data等

1. 资源加载错误

使用window.addEventListener(‘error’)捕获，window.onerror捕获不到资源加载错误。

window.addEventLister(‘error’)捕获到的错误可以通过target?.src || target?.href 区分是资源加载错误还是js运行错误。

1. 接口错误

所有的http请求都是基于xmlHttpRequest或者fetch封装的，所以要捕获全局的接口错误，方法就是封装xmlHttpRequest或者fetch。