# 途中发现的知识点

Object.key(obj) 返回可以枚举的属性名(不包含symbol)

Object.getOwnPropertySymbols() 返回所有symbol属性（不包括父辈原型链上的）

Object.getOwnPropertyNames() //返回所有属性（不包括symbol属性）

Object.values() //返回可以枚举的值（不包括symbol）

Object.is(value1,value2) //比较两个值是否全等

NaN

对于数组

for of 迭代的是 有索引的

for in 枚举的是所有属性（包括数组属性）

**解构放入新对象，新数组**，相当于

var obj={}

var newObj={...obj}

newObj[i]=obj[i]

**如果obj[i]是复杂对象，实际上还是共享的，修改其属性会引发另一个对象变化。**

数组复制与去重复制

function toArr(items){

return [...new Set(items)];

}

function getArr(items){

return [...items]

}

var a=getArr([1,2,2,2,2,5,6]);

console.log(a);

var b=toArr([1,2,2,2,2,5,6]);

console.log(b);

变量赋值

function go(a,b,c,d){

if(arguments.length<4){

[c,d]=[b,c];

}

console.log(c,d);

}

ES6语法

# var 的Hoisting机制

## 块级作用域（词法作用域）绑定

let（块级声明）

块级作用域在

函数内部

{} 内部

const声明

const num =0;

必须初始化。

const 不允许修改绑定，但允许修改值。也就是说声明值：

可以修改对象的属性值。

注意：

const let没有变量提升

因为在js引擎在扫描代码时，会将const和let放入到TDZ（temporal dead zone）临时死区中去。当执行过声明后才会从TDZ中移除。

全局作用域中let const

不会覆盖全局变量，即是不会绑定在全局对象下（如window），而是会覆盖原来的变量，而对Windows不产生影响。

如果不想在全局下创建属性，用const let最安全

而在irame，跨windows调用是用var

## 字符串和正则表达式

UTF-16

每16个序列是一个编码单元

.length chartAt()就是这样实现的。

216  个字符可以用16位编码表示，但是超过的就要用到辅助平面。（代理对）

共32位

## 一些字符串新方法

chartCodeAt()

formCodePoint()

normailze()

includes() //是否包含文本，true|false

startsWith() //

endsWith() //

上面3个都有连个参数，一个match，一个开始位置

repeat（） //重复字符串返回

## 模板字面量

多行字符串

反撇号

var str=`aaaa

bbb`;

或者

var str=`aaa\nbbbb`;

引用模板

var str2=`ggggg${str}`;

正则

u修饰符（有编码单元模式切换为字符模式）

function hasRegExpU(){

try{

var pattern = new RegExp('.','u');

return true;

}catch(ex){

return false;

}

}

y修饰符（lastIndex相关 实例属性stricky相关）

# 函数

## 默认参数

function test(a,b=1000,c=function(){

alert('test');

}){

console.log(a);

console.log(b);

c();

}

默认参数，当不传参数或者传入undefined，则默认参数生效。

默认参数表达式

function getA() {

return 50;

}

function add(a,b=getA()){

return a+b;

}

console.log(add(10));

默认参数可以用表达式表示。

也可以调用前面的参数

function add(a,b=a){

return a+b;

}

**注意：**arguments对象的行为有多不同

参数对象为传入的实参对象

ES5

非严格模式

改变参数变量，会导致参数对象改变

严格模式

改变参数变量，参数对象不会改变

ES6

参数对象与参数变量永久分离，

任何情况下：改变参数变量，参数对象不会改变

参数的临时死区（TDZ）

参数在初始化之前不能被引用，否则报错

当调用函数是，会通过参数或者默认值初始化该参数

## 不定参数

function a(a,...b){

}

参数b为不定参数，是一个数组，表示后来传入的参数数组

对于无法确定参数的数量（针对无命名参数）。

注意：

不定参数不能使用在对象字面量的setter中

因为setter中的参数有且只能有一个

不定参数对arguments无影响

**增强的Function构造函数（但我仍然不用）**

## 函数的name属性

具名函数

.name为函数名

函数表达式

.name为变量名

function a(){

}

function b(){

}

var c=function(){

}

var d=function test(){

}

var e=function(){

}

console.log(e.bind().name) bound e 绑定函数

console.log((new Function()).name) anonymous 匿名

**注意**

JavaScript函数内部有两种不同的内部方法

[[call]] [[construct]]

函数调用，执行call

函数构造，执行construct

判断函数是被调用还是构造

ES5中可以用

instanceof

var Person=function(){

if (this instanceof Person) {

console.log('这里是构造');

}else{

console.log('这里是调用');

}

}

var p=new Person();

Person();

Person.call(p);

但是使用call apply时，这种方法失效

ES6中，在函数内部用new.target

function Go(){

if (typeof new.target !== 'undefined') {

console.log('这里是构造');

}else{

console.log('这里是调用');

}

}

var p=new Go();

Go();

Go.call(p);

## 块级函数

在严格模式下，在块级区域声明的函数只会提升到当前块级上

在非严格模式下，在块级区域声明的函数会提升都全局上

## 箭头函数（=>）

与一般函数区别

1. 没有this,super,arguments,new.target绑定，箭头函数中的这些值都由外围最近一层非箭头函数决定。
2. 不能通过new方法调用，箭头函数没有[[constructor]]方法，不能构造。
3. 没有原型属性 Protoytpe
4. 不能改变this的绑定（函数内部的this不可改变，在函数逇生命周期中始终一致）
5. 没有arguments对象
6. 不支持重复的命名参数

**语法**

let peron = value => value

let person = (value1,value2) => v**alue1+ valu**e2

let person = () => ‘hehe’

let person = (value1,value2) => {

return v**alue1/ valu**e2

}

**空函数**

let person=()=>{}

**返回json**

let person = (value1,value2) => ({a:value1,b:value2})

**立即执行**

let persoon = (num)=>{

return{

getNum:function(){  
 return num;

}

}

}(10)

**没有this绑定**

js的函数内部的this绑定是根据函数调用的上下文环境而改变，这有时候会引发错误。

而箭头

## 尾调用优化

尾调用：指的是函数作为另一个函数的最后一条语句被执行。

function do(){

return gg(); //尾调用

}

在函数的最后的调用会新开一个栈帧（stack frame）,将其推入调用栈表示函数调用。

然而在循环调用时，每一个未用完的栈帧都会被保存在内存中，当调用栈过大（甚至爆帧）时，引发程序问题。

es6中会重用当前栈帧，而不会再新建栈帧（严格模式）

满足尾优化条件：

1. 尾调用不访问当前栈帧的变量（也就是说不是一个闭包）
2. 尾调用结果为函数的返回值
3. 尾调用时最后一条语句

在递归中可能优化明显。

# 展开运算符

**传实参**

let arr=[20,10,10,10,];

Math.max(...arr);

原来

Math.max.apply(Math,arr));

# 扩展对象的功能

## 对象的类别

普通对象

特异对象

标准对象

内建对象

## 对象字面量语法扩展

属性简写

当属性与当前域变量同名时，可以简写。

var name=’tom’;

{

name:name

}

{

name

}

对象方法的简写

{

say(){

}

}

字符串作为键值

person[‘first name’]=’tom’;

在es6中可以用计算的属性名作为键值

var lastName=’last name’

{

[lastName]:’tom’

}

var name=’ name’;

{

[‘last’ + name]:’jim’

}

## 新增方法

**Objiect.is()方法**

===避免了 == 触发的强制类型转换。

弥补全等符的不准确运算

如（+0 ， -0 在JavaScript中是不同实体，但是全等会将认为这两个相等）

NaN === NaN 返回false

**Object.assign()**

实现extend（） mixin( ) ---混合

Object.assign(obj,{a},{b});

而访问访问器属性会直接转换为值。

注意：

es6严格模式下，对象的重复属性，会取最后一个属性，而不是报错。

es6定义了对象属性的枚举顺序

    var obj={

        b:1,

        8:1,

        g:1,

        a:1,

        c:1,

        1:1,

        6:1,

        2:1,

        4:1,

        k:1

    }

    obj.m=10;

    console.log(Object.getOwnPropertyNames(obj).join(''));

12468bgackm

所有的数字排在前面，升序排列

所有的字符串排在字符串后面，按照对象中的顺序排列。

所有symbol键按按照对象中的顺序排列

注意：

for in Object.keys() JSON.stringify() 使用同一枚举顺序，他们的枚举顺序不确定。

增强对象原型

改变对象的原型---在实例化之后

    let person={

        getName(){

            return 'tom'

        }

    }

    let friend = Object.create(person);

    console.log(friend.getName());

    let dog={

        getName(){

            return 'laifu'

        }

    }

    Object.setPrototypeOf(friend,dog);

    console.log(friend.getName())

对象原型储存在[[prototype]]中，调用Object.setPrototypeOf(obj,prototypeObj)改变其中的值。

Object.getPrototypeOf(obj) 获取原型对象

**super**

super的

## 正式的方法定义

es6以前，并未正式定义‘方法’概念。

对象内的方法会有一个内部的[[HomeObject]]属性

# 解构

含义：打破数据解构（对象字面量，数组），将其拆分为更小部分的过程。

对象字面量

let node={

type:'hehe',

name:'jiji'

}

let {type , name} =node;

用一个对象字面量来解构字面量

赋值

let node={

type:'hehe',

name:'jiji'

}，

type=”add”

({type , name} =node);

可以设置默认值

let node={

type:'add',

name:'gg'

}

let {type,name,value=true} =node;

console.log(type,name,value);

非同名局部变量赋值

let node={

type:'add',

name:'gg'

}

let {type:a,name:b,value=true} =node;

console.log(a,b,value);

注意：变量在右，值在左。

嵌套对象解构

let node={

type:'add',

name:'gg',

tese:{

tests:'a'

}

}

let {tese:{test:gg=’bb’}}=node;

console.log(gg);

注意：必须要初始化，也就是赋值（无论是var let const）,否则报错

赋值必须要括起来，否则报错（JavaScript将{}当做一个代码块，语法规定，代码块不能出现在赋值语句左侧，加小括号可以将代码块转化为一个表达式，实现解构赋值）。

读取null，undefined会报错。

声名对象中没有的变量，其值为undefined

数组解构

用数组字面量声明

var arr=['blue','green','red'];

var [,,c]=arr;

console.log(c);

赋值

var arr=['blue','green','red'],a=1,b=2,c=3;

[a,b,c]=arr;

console.log(a);

变量的值交换

var a=10,b=20;

[a,b]=[b,a];

console.log(a,b);

默认值

[a=10,b,c]=arr;

注意

读取null,undefined会报错

嵌套数组解构

var arr=['red',['blue','black'],'green'];

var [,[,a]]=arr;

console.log(a);

不定元素

var arr=['red',['blue','black'],'green',{name:"tom",years:12}];

var [f,...last]=arr;

console.log(f);

console.log(last);

可以用来复制数组

var 【...myarr】=arr;

嵌套数组+对象字面量

var arr=['red',['blue','black'],'green',{name:"tom",years:12}];

var [,[,a],,{name:n,years:y,dom:z=40}]=arr;

console.log(a);

console.log(n);

console.log(y);

console.log(z);

解构参数

意义

定义函数的对象参数时，可以确定对象参数中每一个属性的含义。

function test(a,b,{name:n,type:t='add'}){

console.log(a);

console.log(b);

console.log(n);

console.log(t);

}

test(10,25,{name:'jim'});

注意：

调用函数时，解构的对象参数不能不传，而且对象中参数值不能对应null，undefined

# Symbol 和 Symbol属性

原始类型： Symbol

var firstName=Symbol('this is first name');

var s=console.log(firstName);

Symbol的描述被储存在内部的[[Description]]属性中，只在打印时显示。

代码无法访问

typeof firstName ‘symbol’

Symbol的使用

person[firstName]=’tom’

perosn={

[firstName] = ‘tom’

}

Symbol 共享体系

var uid1=Symbol.for('uidjj')

var obj={name:'ite',[uid1]:'lala'};

function test(){

var uid2=Symbol.for('uidjj');

console.log(obj[uid2]);

}

console.log(obj[uid1]);

test();

Symbol.for第一次调用：创建并记录这个symbol

第二次调用：从Symbol全局中检索这个Symbol

检索symbol在全局的key

Symbol.keyFor(uid2)

Symbol与类型强制转换

由于其他类型没有雨Symbol逻辑等价的值，因而它的使用不会很灵活，无法进行强制类型转换。

# Set集合与Map集合

## Set集合

是一种无重复元素的列表

**var set=new Set();**

**set.add(5);**

**var set = new Set([1,2,3]);**

不会进行强制类型转化，内部调用Object.is()检测两个值是否相等。相等会过滤掉重复值。

对于添加的对象，如果共享相同，会去重；如果不相同，他们彼此保持独立。

**set.zise**

**set.has(5) true**

**set.delete(5)**

**set的forEach循环**

第一个与第二参数都是元素，第三个是set集合本身

**set转化为数组**

let set =new Set([1,5,6]);

array=[...set];

**弱引用集合 weak Set**

一般的set是强引用的集合，只要储存在其中的引用存在，垃圾回收机制就不能释放该对象的内存空间。也就是说直接置空set中的对象元素，实际上这个对象的内存没有被释放。

这也是一种内存泄漏。

--🡪 这里说的强引用，举个例子：

let set = new Set(),

key = {};

set.add(key);

console.log(set.size); // 1

// 取消原始引用

key = null;

console.log(set.size); // 1

// 重新获得原始引用

key = [...set][0];

这里不会释放该对象。

为解决这个问题，引入weak Set，

weak set只存储对象的弱引用，并且不可以存储原始值（不能存数字，字符串，undefined，null）

拥有 add delete has方法，且这三个方法只能传对象，否则报错

weak set集合不能迭代，不能用于for of

不支持 forEach,不暴露任何迭代器（如keys，values）

没有size属性

## Map集合

包好多组键值对的有序列表，集合中每个元素分别存放着可访问的键名和它对应的值（常用来缓存频繁取用的数据）

ES6中的map集合的键名和键值支持所有数据类型。

键名的等价性通过调用Object.is()实现

**ES5中**

可以用对象来模仿set集合和map集合

var set= new Object(null);

set.mytest=true;

if(set.mytest){

//set集合汇总mytest元素存在

}

这里有一个问题，这种方法无法模仿数字类型的元素

set[5]=true;

实际上5会被转为‘5’。

# 迭代器（Iterator）与生成器(Generator)

## 迭代器

迭代器解决的问题：

用循环语句迭代数据，必须要初始化一个变量来记录每一次迭代在数据集合中的位置。

迭代器通过用迭代器对象返回过程中集合的每一个元素，从而简化数据操作

迭代器是什么

迭代器是一种特殊对象，它具有一些专门为迭代过程设计的专有接口。

迭代器对象有一个next（）方法，每次调用都返回一个结果对象，结果对象有两个属性

value : 表示下一个将要返回的值

done ： 【Boolean】 true：没有更多要返回的值

ES5的迭代器

function createIterator(items){

var i=0;

return {

next:function(){

var done = (i >= items.length);

var value = !done ? items[i++] :undefined;

return {

done:done,

value:value

};

}

}

}

console.log(iterator.next())

console.log(iterator.next())

console.log(iterator.next())

console.log(iterator.next())

## 生成器

生成器：是一种返回迭代器的函数，函数名前面加一个\* ，表示这是一个生成器函数。

function \*createIterator(){

yield 2;

yield 4;

yield 1;

}

var iterator= createIterator();

console.log(iterator.next().value);

console.log(iterator.next().value);

console.log(iterator.next().value);

生成器函数的调用，最终返回的是创建好的迭代器。

生成器函数最有趣的部分是：每执行一条yield语句后函数就会自动停止执行。

yield关键字只能在生成器函数中使用 ，即是是生成器内部的函数也不行。

也可以用函数表达式创建生成器

let createIterator= function \*(){

yield 2;

yield 4;

yield 1;

}

**可迭代对象具有 Symbol.iterator属性**，是一种与迭代器密切相关的对象。

Symbol.iterator指定特定函数以返回一个作用于附属对象的迭代器。

而由于生成器默认会为Symbol.iterator属性赋值，因此所有的通过生成器创建的迭代器都是可迭代对象。

**而for of 循环通过调用数组的Symbol.iterator**方法获取迭代器（由JavaScript引擎背后实现），其返回的对象的value被存储在变量中。当done为true时循环退出。

**访问默认迭代器**

var arr=[10,50,10,1,5,8];

let iterator =arr[Symbol.iterator]();

console.log(iterator.next())

console.log(iterator.next())

console.log(iterator.next())

console.log(iterator.next())

判断对象是否为迭代对象

function isIterable(obj){

return typeof obj[Symbol.iterator] === 'function';

}

console.log(isIterable([1,8,2,6]));

console.log(isIterable({a:'name'}));

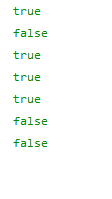
console.log(isIterable('aaaa'));

console.log(isIterable(new Map()));

console.log(isIterable(new Set()));

console.log(isIterable(new WeakMap()));

console.log(isIterable(new WeakSet()));



创建可迭代对象。

默认情况下，自定义的对象都是不可迭代的对象。

给对象的Symbol.iterator 属性提添加一个生成器，则可以变成可迭代对象。

let collection ={

items:[],

\*[Symbol.iterator](){

for(let item of this.items){

yield item;

}

}

}

collection.items.push(2);

collection.items.push(9);

collection.items.push(21);

collection.items.push(12);

for(let i of collection){

console.log(i)

}

评论：好像没有什么屌用，items本来就是可迭代的。

## 内建迭代器

ES6为许多内建类型提供了内建迭代器

**集合对象迭代器**

entries()返回一个迭代器，其值为多个值对

即是值对数组 如【0，‘aa’】

values()返回一个迭代器，其值为集合的值

var set = new Set(['green','blue','red']);

for(let i of set.values()){

console.log(i)

}

keys() 返回一个迭代器，其值为集合中多有键名

**默认迭代器**

每个集合类型都有一个默认的迭代器，在for - of 循环中，如果没有显示的指定则使用默认的迭代器。

**数组, Set 集合默认**是values（）迭代

**Map**则是 entries() 迭代

对于map，for- of 可以解构

let map = new Map()

map.set('title','ismap')

map.set('content','red')

map.set('type','gg')

for(let **[key,value]** of map){

console.log(key)

console.log(value)

}

字符串迭代器

自es5后，就可以通过str[i]这种形式来访问字符串中的字符

但是由于双字节字符被视作两个独立的编码单元，导致双字节字符会被当做两个空格。

而es6后，全面支持Unicode

NodeList 迭代器

DOM标准中有一个NodeList类型，  
 ES6 中添加了默认迭代器。

## 展开运算符和可迭代对象

展开运算符，可以操作所有可迭代对象，并根据默认迭代器来选取要引用的值，从迭代器读取所有值。

## 高级迭代器功能

**给迭代器传参**

可以在next （param）调用中传参

第一次调用next（）之前不会调用任何yeild语句，也就是说第一次传参没有意义。

之后调用next()，会将之前调用的yied赋值为这个参数

function \*createIterator(){

let first = yield 1;

let second = yield first +2 ;

yield second + 3

}

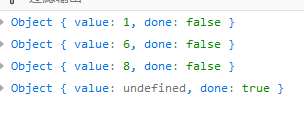
let iterator = createIterator();

console.log(iterator.next())

console.log(iterator.next(4))

console.log(iterator.next(5))

console.log(iterator.next())



**迭代器中抛出错误**

**生成器返回语句**

由于生成器也是函数，因而其也有返回值。他被执行时，可以返回一个值。done也会变为true：表示迭代完成，

注意: 展开运算符和 for-of会直接忽略通过return 返回的任何值。只要done一变为true，就立即停止读取其他值。（就像 : 实际上一般迭代器，到最后一次迭代，没有yield 对应，得到value为undefined，done为false，这个undefined不会被读取）

**委托生成器**

生成器可以合并

合并的生成器中的上一个生成器的返回值（通过return返回的值）不会被返回。

**异步任务执行**

由于生成器支持在函数中暂停代码执行，因而可以深入挖掘异步处理的更过用法

传统方式是用回调实现：任务执行完毕后 调用回调函数。但是对于需要嵌套回调 或者序列化一系列的异步操作，会很复杂。

## yield

yield可以直接用于字符串 yield \* “abc” 直接访问字符串的默认迭代器

注意：

es6中：

所有的集合对象（数组，set集合，map集合）和字符串都是可迭代对象。

# 类

class PersonClass {

constructor(name) {

this.name = name

}

sayName(){

console.log(this.name)

}

}

let person = new PersonClass('niconi')

person.sayName()

console.log(person instanceof PersonClass)

console.log(person instanceof Object)

console.log(typeof PersonClass)

console.log(typeof PersonClass.prototype.sayName)

## 构造函数：

constructor

这实际上是一个语法糖，typeof PersonClass 是 function。因而PersonClass实际上是创建一个具有构造函数方法行为的函数。

## 类表达式

类可以被写出表达式

var PersonClasss = class{

}

匿名表达式

命名表达式

注意：

与函数不同的，类属性可以被赋予新值。PersonClass.prototype 就是一个只读的类属性。

## 与原型链类的区别

1.函数声明可以提升，而类声明与let声明类似，不能提升，真正执行声明之前，它们会一直存在于临时死区。

2.类声明中所有代码自动运行于严格模式，而且无法强行让代码脱离严格模式。

3.原型链类中，需要通过Object.defineProperty()手动指定某个方法为不可枚举。而在类中，所有方法都不可枚举

4.每个类都有一个[[Construct]]的内部方法，通过关键字 new 调用那些不含[[Construct]]的方法会导致程序抛错

5.使用非new 的方式调用类的构造函数会导致抛出错误

6.在类中修改类名会导致程序报错（而在外部可以改变）

## 直接调用类

class PersonClass {

constructor(name) {

this.name = name

}

sayName(){

console.log(this.name)

}

}（‘tom’）

## 一等公民的类

一等公民:可以传入函数，可以被函数返回，可以赋值给变量的值。

## 类的访问器属性

类支持直接在原型上定义访问器属性。

class p {

get html(){}

set html(){}

}

# 数组功能

## 创建数组

new Array()有一些怪异的行为

var arr= new Array(2) length [0] [1]

2 undefined undefined

var arr = new Array(2,3) 2 2 3

var arr = new Array(‘2’) 1 ‘2’

new Array(‘2’,3) 2 ‘2’ 3

**Array.of()**

而Array.of()无论传入什么，都会直接返回一个包含这些函数的数组

**Array.from()**

将类数组对象转换为数组

ES5的做法

Array.prototype.slice.call(arrayLike)

----> 由于slice只需要数值型索引和length 就可以正常运行。因而可以将类数组对象转换为数组

ES6做法

Array.from(arguments)

可以传一个回调，映射转换

Array.from(a,(value) => value + 1)

**三个参数**

obj cb target

第三个参数为cb中的this绑定值

**可以转换可迭代对象**

## 数组中搜索

**带条件的搜索**

**find()**

两个参数

回调 this绑定

**findIndex()**

返回索引

这两个搜索只搜索一个符合值，就返回

**搜索固定值（ES5）**

indexOf()

lastIndexOf()

## 填充数组

**fill()**

参数

target start end

将索引 >= start <end 的项都转换为target

不指定end 则end为arr.length

start , end 为负值

则 start + arr.length end+ arr.length为真实的位置

**copyWithin()**

参数

fillStart copyStart num

开始填充的位置 开始复制位置 限制被重写元素的数量

也可以写负值

## 定型数组

定义：用于处理数值类型数据的专用数组。

最早是在webGL中使用的，webGL是openGL ES 2.0 的移植版，在web页面中通过canvas元素来呈现它。定型数组也一并移植过来，它可以为JavaScript提供快速的按位运算。

需要定型数组的原因

JavaScript中数字是64位浮点格式存储的，并按需转换为32位，算数运算非常慢。

ES6中引入定型数组来解决这个问题

所谓定型数组，就是任何数组转换为一个包含数字比特的数组，随后既可以通过JavaScript数组方法来进一步处理。

# promise

解决的问题

当更多程序开始使用一步编程时，事件和回调函数却不能满足开发者想要做的所有事。

背景

JavaScript是基于单线程事件循环的概念而构建的，同一时刻只运行一个代码块在执行，与之相反的是像java 和 c++允许多个代码块同时执行。对于基于线程的软件，多个代码块同时访问并改变状态时，程序很难维护并保证状态不会出错。

而JavaScript引擎同一时刻只能执行一个代码块，代码都被放在一个任务队列。

每当一段代码即将执行时，就会被添加入任务队列（如事件函数）

**事件模型**（针对响应用户交互和完成类似的低频功能）

**回调模式**

**回调的问题**

promise相当于异步操作结果的占位符

**promise的生命周期**

进行中 pending

成功完成 fulfilled

未成功 rejected

内部属性 [[PromiseState]]用来表示Promise 的三种状态

而这个属性不能通过编程的方式来检测，只有当Promise的状态改变时，通过 then（）方法来采取特定的行动

.then(cb,errorCb)

实现了then()方法的对象，称为thenable对象。所有的pormise都是thenable对象，但是并非所有thenable对象都是promise

.catch(errorCb)

reject()失败通过

resolve()成功通过

new Promise(function(resolve,reject){

fs.readFile(filename,{},function(err,contents){

if(err){

reject(err)

}

resolve(contents)

})

})

Promise对象用于传递异步消息，代表了某个未来才能知道结构的事件（通常为一个异步的操作）。为这个事件提供统一API，供进一步处理

1.1 Promise对象的两个特点

对象不受外界影响，Promise对象代表一个异步操作，有三个状态：pending(进行中)

resolved(已完成) rejected(已失败)。只有异步操作的结构能够决定其状态，任何其他操作都无法改变这个状态。

一旦状态改变，就不会再变。且只有两个可能（pending到resolved ， pending到rejected）

## 类方法

**Promise.all(iterable)**

返回一个Promise实例，此实例在iterable参数内所有的promise都完成或参数中不包含promise时 resolve。

iterable : 一个可迭代的对象 如Array， String

如果传入的是：

空的可迭代的对象，直接返回一个resolve状态的promise

非空，不包括任何promise，则返回一个异步resolve的promise

包括promise 时，返回pending的promise

Promise.race(iterable)

返回一个promise ，其实resovel还是reject 取决于第一个完成的promise的状态。

Promise.resolve(value)

返回一个给定值解析后的promise对象。如果这个值是个thenable（即带有then方法），返回的promise会跟随这个thenalbe对象。采用它的最终状态（resolveed/rejected/pending/settled）.否则返回resolve 的promise

Promise.reject(reason)

返回一个被拒绝的 promise

## 实例方法

Promise.prototype.then(onFulfilled, onRejected)

then中的回调函数的 返回值

1 一个具体值

then返回的promise将会变为resovle，并且将返回的值作为resolve的参数

1. 一个错误

返回的promise 为 reject ，并将错误作为reject 参数

1. 一个resolve 的promise

返回的promise也为resolve，并且参数相同

1. 一个 reject 的promise

返回的promise 也为 reject，并且参数相同

1. 一个 pending 的promise

返回的promise 也是 pending，并且它的终态也与then中返回的promise的终态相同，参数相同。

Promise.prototype.finally(onFulfilled, onRejected)

Promise.prototype.catch()

与 Promise.prototype.then(undefined, onRejected)相同

# async await

/pages/feature/asyncAndawait/asyncAndawait.html

async声明一个返回 AsyncFunction 对象的异步函数

实际上是Generator的语法糖，只是将 \* 替换成async， 将yield替换成了await。

generator会暂停程序，直到获取值为止。这就为异步操作提供了一种可能

而async函数对generator进行了改进：

1. 内置执行器

内置执行器，不需要next()调用

②更好的语义

async声明异步操作

await表示后面的表达式需要等待结果

③更广的使用性

await后可以是任何值或promise

④返回promise

# 代理（proxy）和反射（feflection）

代理proxy是一种可以拦截并改变底层JavaScript引擎操作的包装器，通过它暴露内部运行的对象，从而让开发者可以创建内建的对象。

new Proxy() 创建一个代替其他目标对象的代理，它虚拟化了目标。

代理拦截JavaScript引擎内部目标的底层对象操作，这些底层操作被拦截后会触发响应特定操作的陷阱函数。

反射API以reflect对象的形式出现，对象中方法的默认特性与相同的底层操作一致，而代理可以覆写这些操作，每一个代理陷阱对应一个命名和参数都相同的Relect方法。

async

async函数的使用方式，直接在普通函数前面加上async，表示这是一个异步函数，在要异步执行的语句前面加上await，表示后面的表达式需要等待。async是Generator的语法糖，相比较Generator函数在以下四方面做了改进

# 模块化

## 导出导入

export

import

导出变量只能在当前文件中被修改，而不能在引入文件中修改（保存）

注意：

Import是静态执行，不能使用表达式和变量!！

动态用法？？