ECMAScript6的新特性

1.块级作用域

块级作用域，又被称为词法作用域，可以再函数内部，或代码块即`{}`内部创建

1. 用`let`声明的变量具有块级作用域，声明的变量不能在外部访问，只能在块级作用域中访问，并且变量没有声明提前也不能重复声明
2. 使用`const`声明的变量成为常量，常量不能再被赋值，也具有块级作用域的的特征,一般使用大写字母来表示常量，常量必须在声明时进行初始化赋值。当对象是一个常量，只是它里面的地址值不允许改动，内部属性等还是可以改动的。
3. 循环中块级绑定

```
//如果不用块级作用域，则要使用闭包
for (var i = 0; i < 10; i++) {
    setTimeout ( (function(i) {
        return function() {
            console.log(i);
        }
    }(i)),2000);
}
//使用let
for (let i = 0; i < 10; i++) {
    setTimeout(function() {
        console.log(i);
    } , 2000);
}
```

4. 暂存性死区：在代码块内使用let声明变量前，该变量都不可用（temporal dead zone简称 TDZ）

```
var tmp = 123;
if(true){
  tmp = "abc";//报错ReferenceError
  let tmp;
}
```

2.函数形参的默认值

可以给具体的值，也可以给表达式，变量，函数调用的返回值，形参的默认值是动态设置，只有在函数调用时才会进行赋值，默认值只有需要时才会赋值，不需要不会自动赋值，undefined相当于没有传值也会使用默认值，但null不会使用默认值。

一个函数使用了参数的默认值，arguments的特性和在es5下的严格模式一致，如果使用了默认值则不能在函数内部 使用`"use strict"`但可以使用全局严格模式。

`function(name = "lisi"age = (10+20)){}`

3.剩余参数和展开（扩展）运算符

剩余参数是一个纯数组，使用`...变量名`来表示，剩余参数必须是形参的最后一个。

展开运算符，除了在函数的形参列表，和解构中的左边，其他地方的`...`都是展开运算符，可以用开展开数组、和类数组。将可以迭代的变量等给展开迭代。

4.箭头函数

箭头函数，就是拉姆达表达式（lambda expression），本质上是一个匿名函数，最普通的箭头函数，（主要用来替换匿名函数）

```
var foo = (a, b) =>{
    // 函数体
    console.log("我是箭头函数", a, b);
}
//当只有一个参数时可以省略（），当函数体只有一行代码是并且这行代码不//能有return函数会自动返回这行代码计算后的返回值，
let foo = a => a * a
```

箭头函数没有自己的this，但会查找使用上一级知道全局环境下的this，回调函数使用箭头函数有最佳体验，可以得到想要的this，箭头函数也没有自己的argument

```
var obj = {
    age: 20,
    foo: function (){

        let f = ()=> console.log(this);
        /*function f(){
            console.log(this);
        }*/

        f();
    },
    foo1 : function (){
        setTimeout(()=>{
            console.log(this);
        }, 1000)
    }
}
 obj.foo();//obj
```

给对面字面量添加方法时不要使用箭头函数,箭头函数没有自己的this，会找到window

```
var age= 100;

var obj = {


    age : 20,
```

```
    foo : ()=>{
        console.log(this);
        console.log(this.age);
    }
}
obj.foo();//window,100

var obj1 = {
    age : 10,
    foo : function (){
        let f = ()=>{
            console.log(this.age);
        }
        f();
    }
}
var obj2 = {
    age : 20,
    foo : obj1.foo
}

obj2.foo();//20
```

箭头函数也会立即执行`var r = (() => console.log("abc"))();`

5.对象功能的扩展

1. 普通对象，特异对象，标准对象，内置对象（标准对象都是内置对象）
2. `{name,age}`找一个同名的变量赋值，找不到就报错，但name不会报错，因为window下有一个name属性值是一个空字符串`""`。

```
function createPerson(name ,age){
  return{
    name : name,
    age : age
  }
}
//简写一、返回和属性名同名的变量赋值
function createPerson(name ,age){
  return{name,age,sex:"男"}
}

var p1 = createPerson("李四",20);//{name:李四,age:20}
var obj = {name}//window下有一个name是一个空字符串，因此会找到一个空字符串。
var obj = {
  say:function(){
    console.log("say");
  }
  speak(){
    console.log("speak");//对象的方法的创建的简写
    }
}
```

3. 动态计算对象字面量的属性

```
var prop = "age";
var i = 0;
function getPropName(){
  return "name" + i++;
}
var obj = {
  [prop]: "李四",//新方法
  [prop + 1] : "李四",
  [getPropName()]:"ddd",
  [prop + 100] : function (){
        console.log("bbb");
    },
   [prop +200](){
      console.log("aaaa");
   }
};
obj[prop2] = "李四"//老办法
console.log(obj);
obj.age200();
obj[prop + 200]();
```

4. 添加方法

- 优先考虑具体的类型，不是所有的都向Object.prototype上添加
- 考虑和具体对象的关系
- 如果和具体的对象的关系不大，考虑添加到构造函数上，而不是添加到构造函数的原型上。

如：`Object.is()`，判断两个值是否相等 ，在绝大部分下都是和 `===` 相同，

不同的地方：

- `Object.is(NaN,NaN)` 结果是true，`===`是false
- Object。is (+0,-0) 结果是false，`===`是true

`Object.assign(obj1,obj2,{sex :"男"},{sex : "女"})`将`obj2,{sex :"男"},{sex : "女"}`的值赋给`obj1`，返回值就是新的`obj1`

```
var obj1 = {};
var obj2 = {};
var obj3 = {};
var target = Object.assign({},obj1,obj2,obj3);//常用用法
//Math中的赋值不允许，window可以因为遍历不了，属性设置的问题
//只能复制允许遍历的属性,复制基本上都是浅复制，对象会复制地址值
var ibj1 = {
  girls : ["a","b"]
}
var target = Object.assign({},obj1);
console.log(target);//{girls:["a","b"]}
obj1.girl.push("c");

console.log(target.girls);//["a","b","c"]
```

```
console.log(obj1.girls === target.girls);//true
```

6.字符串的功能的增强

6.1查找字符串

1. `includes()`方法

   如果给定文本存在与字符串的任意位置是返回true，否则false

2. `startsWith()`方法

   如果给定文本在字符串开头时返回true，否则false

3. `endsWIth()`方法

   如果给定文本在字符串结尾时返回true否则false

这三个方法都接收两个参数:需要搜索的文本和可选的起始索引值,，如果没有第二个参数的传入则，前两个方法从头开始查找，end从最后开始，如果有值，则以这个值作为结尾向前查找，其他两个则正常依次为开始位置向后查找

`indexOf()`,找到的是索引。

6.2 `repeat()`

字符串进行重复，`var s = "我"; var ss =s.repeat(3);console.log(ss)//我我我`重复三次返回重复之后的字符串。

6.3字符串模板

```
//以前的时候
var s = "abc \
bcd \
哈哈哈 \
";
//如果要换行\n
//新的反引号``
var s = `alkjalfjlksa
as;kjfdalkadf;kj
fjaskljfal`;
//原样输出，不会改变格式
//字符串模板替换
var a = 4,b = 5;
console.log(`a的值是：${a}，b的值是：${b}`)
//${},$外的空格影响字符串，但内部的不会
//UC浏览器老版的不支持字符串模板，新版未知
var mytag;
var c = mytag`alfjfkj${a}`;//is no a function
function mytag(arr,v1,...vs){//插入的值，可以多个也可以剩余参数
  cosole、log(v1);
  cosole、log(arr);
  return "aaaa";

}
```

```
var c = mytag`alfjfkj${a}`;//aaaa
//得到的字符串是模板标签的返回值，模板标签的本质是一个函数
```

```
//模板标签存在的意义是对模板进行处理
function getAge(){
  return 20;
}

function myTag(arr,...vs){
  var ss = "";
  for(var i = 0; i < arr.length;i++){
    ss +=arr[i];
    vs[i]&&(ss += vs[i]);
  }
  ss +="欢迎，欢迎";
  return ss;
}
var s = myTag`我的名字叫：${name}，我的年龄是：${getAge()}。`
console.log(s);//我的名字叫： ，我的年龄是：20欢迎，欢迎
```

7. `deconstruct`,解构/析构

解构的时候可以使用剩余参数。

```
 let [a,b,...c] = [10,12,30,40,50,60,70];
//c是一个保存剩余参数的数组
let {a,b,...c} = {a:10,b:20,d:20,r:300};
//c是一个对象
```

1. 对象解构

   对象中的属性的解构`let {name,age} = {name " lisi",age:20}`相当与声明两个变量，找到在对象中属性名相同的值进行赋值。

   结构表达式`{naem,age} =obj`,当想要在声明后，重新赋值，使用解构表达式，但必须加`({name,age} = obj)`,默认值`let {name = "zhang-san",age} = obj`。

   当属性名不同时使用异名解构`let {name:n = "zhang-san",age:a} = obj`

2. 数组的解构

```
let a = 3,b = 4;
[a,b] = [b,a];//数组的解构，交换a，b的值
console.log(a);//4
console.log(b);//3

let arr = [1,2,3,4,5,6,7,8];
//非顺序赋值
let [,first,second,,third] = arr;
console.log(first,second,third);
```

交换的几种方法

```
var a = 10,b = 20;
//1.使用第三方变量
var c= a;
a = b;
b = c;
//使用和差的办法，只能交换数字
a = a + b;
b = a - b;
a = a - b;
//使用异或的办法，只能用于数字
a = a ^ b;
b = a ^ b;
a = a ^ b;
//使用数组结构，任何类型都可用
[a,b] = [b,a];
```

3. Array.from()

   将类数组转换为真正的数组，并返回数组`let arr = Array.from("adddffgg");`或者`let arr = [..."abc"];`

4. Array.of(element,element,element,...)

   使用传入的参数创建新的数组，`let arr = Array.of(10,2,3,4);let arr1 = new Array(10,2,3,4);`构造函数可以省略new，构造函数如果传入一个数字，则会变为数组的长度，创建n个undefined，但Array.of(10)不会。

5. Array.prototype.fill(n)

   把数组中的每一个位置，用1填充`arr.fill(1)`。

6. Array.prototype.find()

   find找到或找不到，结果是一个布尔值Boolean`arr.find(callBack（元素,下标，原数组）[]),indexOf()使用'==='判断`，回调函数判断是否相等，是返回true，否返回false，只会返回第一个满足的元素。

   `arr.findIndex()`找到的是下标

7. Array.prototype.entries()

```
let arr = [1,2,3,4] ;
var it = arr.entries();//迭代器
console.log(it.next())
```

8. `Array.prototype.copyWithin(target,start,end)`拷贝数组的值

```
[1,2,3,4,5].copyWithin(-2);//[1,2,3,1,2]
[1,2,3,4,5].copyWithin(0.3);//[4,5,3,4,5]
[1,2,3,4,5]/copyWithin(0,3,4);//[4,2,3,4,5]
//参数一copy的值从那个位置开始填充，参数二填充取值的开始的位置，结束的位置，前闭后开，操纵的原数组，返回值也是原数组，不会更改数组的长度
```

9. `arr.isArray()`判断是否是数组。

```
//判断数组的方法
arr instanceof Array
Object.prototype.tostring.call(arr)
Array.isArray(arr)
```

10. 修改数组的方法

    pop,push,sort,splice,unshift,shift,等

11. 不修改数组的方法

    slice,toSource,indexOf,等

12. 迭代方法

    entries，every，filter等

[参考地址][https://devdocs.io/javascript/global_objects/array]

8.symbol(符号)新的数据类型

基本数据类型，主要用来做对象的属性名

1. 创建symbol类型的数据：（没有字面量）

   `let s1 = Symbol();`，用这种方式多次获取的Symbol，不相等。

2. 区分symbol变量

   `let s1 = Symbol("标识符"),s2 = Symbol("b");`,

3. 获取的其他方法

```
var s3 = Symbol.for("abc");//使用Symbol。for(key),根据一个key获取Symbol
var s4 = Symbol.for("abc");
s4 ===s3 //true
//先在全局的window上找一个键为“abc”的Symbol如果，没有就创建一个注册到全局环境中，如果有就直接拿到
//Symbol.keyFor(Symbol对象);拿到key
console.log(Synbol.keyFor(s3));
```

4. 遍历

```
        var obj = {

            [Symbol()] : "aaa",
```

```
            age : 20,
            [Symbol()] : "bbb"
        }

        console.log(Reflect.ownKeys(obj));
        console.log(Symbol.iterator);

         var arr = Object.getOwnPropertySymbols(obj);
         console.log(arr);
         var arr1 = Object.getOwnPropertyNames(obj);
         console.log(arr1);
        for(var propName in obj){
            console.log(propName);
}
```

9.集合（set，collections）

set中存储的元素不允许重复，set集合中元素没有下标。

`let set = new Set()`创建一个set集合，

`set.size`集合的长度length

```
let set = new Set();
//添加set集合元素
set.add(1);
set.add("qbs");
set.add(false);
//删除元素
set.delete(1);
//集合长度的获取
console.log(set.size);
//清空集合中的元素
set.clear();
//查查看有没有元素
set.has(ele)//结果是true 或者 false
//获取元素,遍历for-each for-of，**注意：for-in和普通的for循环都不能遍历set集合**
set.forEach(function (value,key,self) {
  console.log(value,key);//都是集合的元素，键和值是一样的
})
for(let v of set){
  console.log(v);//元素
}
//迭代器可以遍历
let it = set[Symbol.iterator]();
console.log(it.next().value);
obj =  it.next();
while(!obj.done){
  console.log(obj.value);
  obj = it.next();
}
//遍历时的顺序是添加的顺序
//到目前的可迭代对象有数组，字符串，Set,Map
```

set集合的初始化

`let set = new Set([1,2,3])//1,2,3,set1 = new Set("abc");//:"a","b,"c`

set集合中的元素的不可重复性

按严格相等（Object.is()）判断，重复时，添加失败。但有两个地方不一样

- **+0和-0**，相等
- **NaN和NaN相等**

undefined和null也可以存储但只能一次。

```
//去重,应该封装方法
let arr = [1,2,3,4,56,1,2,4,56,76,,8];
let set = new Set(arr);
arr.splice(0,arr.length);//arr.length = 0;
arr.push(...set);
//接受一个数组，去除重复元素（在原数组中操作），可以再返回这个数组
function removeDuplication(arr){
    let set = new Set(arr);
  arr.splice(0,arr.length);//arr.length = 0;
    arr.push(...set);
  return arr;
}
//接受一个数组，去除重复元素返回新数组
 function removeDuplicationAndNewArray(arr) {
      return [...new Set(arr)];
 }
```

10.map（字典，映射，地图）数据类型

map中存储的都是键值对，key-value pair。对象中的键（属性名）必须是字符串或Symbol类型，值（属性名）。map中的键可以是任何类型，没有类型限制。

在map中键和值是一种多对一的映射关系，：一个键只能有一个值，但多个键可以指向一个值

key不允许重复，值没有任何限制，也可以重复。

- map.size返回键值对的个数

  `let map = new Map();console.log(map.size);`

- map.set(键key，值value)

  给map集合添加键值对。

  当key重复了则会使用新的值去替换旧值。

  set的返回值是map集合自己，可以链式调用。

  undefined也可以做键，null也可以当键，NaN也可以，{}，{}是不同的键

- map.get(key)

  根据指定的key的值来获取值，get{}),不能获取到{}的键的值

- map.delete(key);

根据指定的key的值，删除key和value

- map.clear();

  清空map集合，清除所有的键值对

- map.has(key)

  根据key的值去找有没有这个键，返回值是一个Boolean

map集合的遍历

遍历的顺序是插入的顺序

1. for-of

```
for(p of map){
  console.log(p);//p是一个数组，[key,value]
}
```

2. for-each

```
map.forEach(function(value,key,self){
  console.log(value,key);
});
```

3. 迭代器

```
let map = new Map([["a",1],["b",2],["c",3]]);
let it = map[Symbol.iterator]();
console.log(it.next().value);
obj =  it.next();
while(!obj.done){
  console.log(obj.value);
  obj = it.next();
}
```

map集合的初始化

map使用二维数组初始化，二维数组中的值是键和值组成的只有两个值得数组

```
let map = new Map([["a",1],["b",2],["c",3]]);
 console.log(map.size);//3
```

11.迭代

两个协议实现了，对象就可以迭代。

- 可迭代协议（iterable）

如果对象要可迭代，则这个对象必须拥有一个属性，这个属性名是通过[Symbol.iterator]来动态计算出来。这个属性的值是一个函数，是一个迭代器生成函数，调用函数就会返回一个迭代器。

- 迭代器协议 迭代器其实是一个对象，但这个对象必须提供一个方法，调用这个迭代器的next（ ）方法的时候，这个方法必须返回一个对象，这个对象必须有两个属性：
  - value：值是迭代出来的数据
  - done：这次迭代是否完成，如果是true表示完成，那么value即使是undefined，如果是false，则value是对象中的值

```
let obj = {
  [Symbol.iterator]:function(){

  }
}
```

自定义迭代器

```
//es5
function createIterator(arr){
  var i = 0;
  return{
    next(){
      let value = arr[i], done = false;
      if(i >= arr.length)done = true;
      i++;
      return{value,done};
    }
  }
}
let it = createIterator();
let obj = it.next();
console.log(obj.done,obj.value)

//纯es5
  var arr = [1,2,3,4,5,6,7,8,8,9];
    function createIterator(arr) {
        var i =0;
        return {
            next: function(){
                let done =false,value = arr[i];
                if(i>=arr.length)done=true;
                i++;
                return{value:value, done:done};
            }
        }
    }
    var it = createIterator(arr);
  console.log(it.next());

//es6的迭代器的语法支持

//在函数名的前面添加一个*，这样的函数就不是一个普通函数，是一个迭代器生成函数了
```

```
//在调用迭代器的生成函数的时候，并不会执行里面的代码，只会返回一个迭代器
//返回的迭代器就满足迭代器协议
//装饰模式
function*createIterator(arr){
  yield:1;
  yield:10;
  yield:100;
  yield:1000;
}
let it = createIterator();
let obj = it.next();//1,false
let it = createIterator();
let obj = it.next();//10,false
let it = createIterator();
let obj = it.next();//100,false
let it = createIterator();
let obj = it.next();//1000,false
//在生成器函数的内部添加yield语句，会在调用next（）的时候，返回yield后的值作为value值,每调用next()一次，从上一次暂停的位置到新的yield暂停
function*createIterator(arr){
  for(var i = 0;i<= arr.length;i++){
    yield arr[i];
  }
}
```

可迭代对象

```
var obj = {
  age : 10,
  name : "李四",
  eat : function (){},
  speak : function (){}
  //[Symbol.iterator]:function *(){}
  *[Symbol.iterator](){
   for(var propName in this){
       if(typeof this[propName == function])continue;
       yield[propName,this[propName]];
       }
   }
}
var it = obj[Symbol.iterator]();
var a = it.next();
//for-of现在有迭代器之后的，就可以使用for-of遍历
```

Object原型添加迭代

```
Object.prototype[Symbol.iterator] = function *(){
   for(var propName in this){
       if(typeof this[propName == function])continue;
       yield[propName,this[propName]];

}
```

```
var obj = {
  age: 1,
  age1: 2,
  age2: 3
}
for(var p of obj){
  console.log(p);
}
var it = obj[Symbol.iterator]();
var a = it.next();
while(!a.done){
  console.log(a);
  a = it.next();
}
```

类

1. es5模拟类

```
function Person(name,age){//类
  this.name = name;
  this.age = age;
}
Person.prototype = {
  construct:person,
  eat :function(){}
}
```

2. es6的类

```
class Person{
  constructor(name,age){
    this.name = name;
    this.age = age;
  }
  eat(){};
  speak(){}
}
var p1 = new Person("lisi",20);
console.log(p1);
```

   - 在类中声明的方法自动放在原型上，原型中的constructor指向class Person

3. 类不会声明提前，具有块级作用域
4. 类中自动处于严格模式

匿名类

```
const Person = class{
  constructor(){};
```

```
  eat(){};
}
//类的自执行
var p1 = new class{
  constructor(name,age){
    this.name = name;
    this.age = age;
  }
}("lisi",30)
//动态获取
var prop = "eat";
class Person{
  constructor(){

  }
  [prop](){//非静态方法，实例方法，方法名动态获取
    console.log("aaa");
  }
  static foo(){//静态方法，也叫类方法，工具方法
    console.log("我是静态方法")；//只能用，类名.静态方法名，来调用，实例不能调用
  }
}
```

原型替换

```
//es5
function Father(name){
  this.name = name ;
}
Father.prototype = {
  constructor : Father,
  eat:function(){}
}
function Son(name,age){
  Father.call(this,name);
  this.age = age
}
Son.prototype = new Father("");
Son.prototype.speak = funciton (){
  console.log("speak",this.name);
}
var s1 = new Son("lisi",20);
s1.speak();
s1.eat();
//es6
class Father{
  constructor(name){
    this.name = name;
  }
  eat(){
    console.log("eat...",this.age)

  }
```

```
}
class Son extends Father{//extends扩展，继承，静态方法也可以继承
  constructor(name,age){
    //首行必须要调用父类的构造函数
    super(name);
    this.age = age;
  }
  speak(){
    console.log("speak...",this.name);
  }
  eat(){
    super.eat();
    console.log("son-eat");
  }
}
var s1 = new Son("张三",20);
console.log(s1);
s1.speak();
s1.eat();
//继承也会将父类的静态方法继承过来
class Father {
        static foo() {
            console.log("弗雷尔卓德之心");
        }
    }

    class Son extends Father {
    }

    Son.foo();//弗雷尔卓德之心
```

迭代器的补充

```
var it = arr.entries();//得到一个迭代器,数组，set集合，map集合都有这个方法
//it.next()迭代出来的是一个对象，value是一个数组，[key,value]数组中键是下标，set中键的值和value的值相同
var it = arr.keys();
var it = arr.values();
//这两个方法返回的迭代器it，next方法迭代出来的的就是键和值，set的键就和值一样，数组键就是下标
var a = [1,2,3,4,5,6,7];
var b = new Set(a);
var it = b.keys();
console.log(it.next())
// {value: 1, done: false}

var it2 = b.values();
console.log(it2.next());
//{value: 1, done: false}

console.log(it2.next());

//{value: 2, done: false}
```

```
console.log(it2.next());
//{value: 3, done: false}

console.log(it2.next());
//{value: 4, done: false}

console.log(it2.next());
//{value: 5, done: false}

console.log(it2.next());
//{value: 6, done: false}

console.log(it2.next());
//{value: 7, done: false}

console.log(it2.next());
//{value: undefined, done: true}

console.log(it2.next());
//{value: undefined, done: true}
```

promise（承诺）

同步和异步，promise主要用来解决异步的问题

异步定时器，回调函数，异步一定是回调函数，回调函数不一定是异步的

promise是一个对象，返回：回调的失败或成功。

promise的生命周期：

- 未定状态unsettled/pending挂起状态：创建承诺到返回结果之间的时间
- 将来会进入**fulfilled（resolved）已决状态**或者**rejected：拒绝状态**

并在之后会一直保持这种状态

```
var promise = new Promise(function(resolve,reject){
  resolve();//已决

});
//promise.catch();
promise.then(function(){},function(){})//成功后的函数，失败后的函数
//new Promise（执行器函数）;
//执行器函数有两个参数都是函数，参数一表示已决，参数二表示拒绝。执行器函数是同步的立即执行，（可以在执行器内执行一些异步的操作）但函数还是同步执行的
```

1. 包装执行函数会立即执行
2. 保证你使用then绑定的这些函数，一定会在当前任务执行后执行then中的函数
3. 如果承诺已决则会调用then的第一个函数，拒绝一定会调用第二个函数

4. 你通过then绑定在这个promise已决或未定之后的所有的操作，可以接受两个参数，但一般会省略第二个，拒绝会抛出异常，多个then一定会按照绑定顺序执行。
5. promise。catch（function (){console.log("已拒绝")}）；拒绝一般使用catch处理

promise中的链式调用

每个then方法调用后会返回一个新的promise，新的promise会与上一个的promise的状态一致，当then中的参数函数的返回值是一个promise对象时，它的返回值就会影响then的返回值的状态。

```
    var img = document.querySelectorAll("img");

     function createPromise(img, src) {
         return new Promise(function (resolve, reject) {
             setTimeout(function () {
                 img && (img.style.width = "300px");
                 resolve();
             }, 500);
             img.src = src;
         });
     }
var i = 0;
createPromise(img[i++], "images/" + i + ".jpg").then(function () {
         return createPromise(img[i++], "images/" + i + ".jpg");
     }).then(function () {
         return createPromise(img[i++], "images/" + i + ".jpg");
     }).then(function () {
         return createPromise(img[i++], "images/" + i + ".jpg");
     }).then(function () {
         return createPromise(img[i++], "images/" + i + ".jpg");
     }).then(function () {
         return createPromise(img[i++], "images/" + i + ".jpg");
     }).then(function () {
         return createPromise(img[i++], "images/" + i + ".jpg");
     }).then(function () {
         return createPromise(img[i++], "images/" + i + ".jpg");
     }).then(function () {
         return createPromise(img[i++], "images/" + i + ".jpg");
     }).catch(function () {
         console.log("执行失败了");
         }).finally(function(){
             console.log("ni hao")
         })
         //改进
         var count = 0, i = 0;
     function thenPromise(p) {
         count++;
         var p1 = p.then(function () {
             return createPromise(img[i++], "images/" + i + ".jpg");
         });
         if (count >= img.length) {
             p1.catch(function () {
                 console.log("执行失败了已经加载了"+ (i - 1) +"张图片了,没有更多的img标签了");
             });
             return;

         }
```

```
        thenPromise(p1);
    }
    thenPromise(createPromise(img[i++], "images/" + i + ".jpg"));
```

Promise.resolve()绑定异步的函数,直接返回一个已决的promise对象

Promise.reject(reason)返回一个拒绝的promise对象

```
    let p1 = Promise.resolve(10);
    p1.then(function(value){
    console.log(value);
    })
    console.log("a")//a,10
```

Promise.all(参数是可迭代对象)，返回值是一个promise对象只有当可迭代对象中的promise对象都已决，他才会已决。，只要有一个拒绝，则返回的对象就是拒绝状态

Promise.race([.....promise对象])，只要有一个已决，就已决。

异步函数

1. 异步函数使用关键字 async
2. 默认情况下, 异步函数总是返回一个 已决的 promise
3. 异步函数的内的代码不会阻塞.
4. await : 等待, 一般是用来等待一个promise的完成
5. 即使内部使用了await方法, 但是promise也会立即返回.
6. await只能用在async函数内. 在外面用会出现语法错误.

```
function createPromise(time){
    return new Promise(function (resolve, reject){
        setTimeout(function (){
            resolve(time);
        }, time)
    })
}

async function foo(){
    console.log("cccc")
    // 等待: 只有这个promsise已决了才会向下执行. a是promise的那个值
    var a = await createPromise(3000);
    console.log(a)
    return "abc";
}

var result = foo();   // resolve("abc")
console.log("bbb")
result.then(function (value){
    console.log(value);
})
async function foo(){

    /*console.log(a);*/
```

```
    throw new Error("错误")
}
foo().catch(function (value){
    console.log("拒绝", value);
})
```