# ECMAScript6

## 关于Node.js

node.js其实可以理解为javascript在服务器上面的运行环境。

我们以前在学习javascript总有一个概念，javascript运行在浏览器的网页里面，它必须依托于网页存在，没有网页就没有它。但是后期人们将javascript 运行在浏览器上面的一个环境`chrome V8`移植到了服务器上面，这样就可以保证我们的js代码能够在本地的服务器上面运行了，而不依赖于浏览器与网页

node.js它只是一个运行平台，不关于任何标准，只要是javascript语言，我都可以在上面运行

## 什么前端需要学Node.js

按照以前的开发思维，前端是不需要学习后期的具体知识，只要掌握一些基本的后端原理即可。但是现在的主流开发方式上面已经趋向于前后端分离式开发，而在这种开发模式下面，后端不负责前的问题，而前端反而需要掌握一些后端的知识，如果数据建模，数据库设计，SQL语句，以及http请求原理，cdn优先等一系列后端知识

在之前的时候后端知识我们都是以`php`为方式来学习的，但是毕竟`php`是一门新的编程语言，它的接受难度与学习成本都比较大，这样做不合适。nodejs推出了以后它完全可以替代`php`语言，必须还可以在我们的JS上面做很多的扩展，我们有很多的插件都是可以运行在nodejs上面的，并且它没有学习一门新的语言，**我们之前JS是可以完全运行在上面的**

node.js还具备一些先天性的优点

1. 非阻涉的IO
2. 使用事件驱动
3. 天生的高并发处理能力
4. 单线程的执行

node.js它只是一个运行平台，在这个平台上面运行是javascript语句，但是一般情况它，它的兼容性非常强（nodejs是装在服务器上面的，服务器的环境是由我们来决定的，不是由客户的浏览器来决定的，所以我们可以实时更新自己服务器的运行环境）

只是因为没有兼容性，所以我们在nodejs上面运行的js代码可以是更高级版本的JS代码。这个时候我们的`ECMAScript6`就可以直接在上面运行（包括后面的ES7,ES8）

**node.js里面只有ECMAScript，没有DOM，也没有BOM**

---

# ECMAScript6

在刚刚开始学习JS的时候，我们的老师都告诉过大家，我们学习的JS的版本是`ECMAScript5.1`，这版本出来非常早，它在设计之初就存在了很多缺点以及不完备的功能，如变量的定义，流程控制，变量的区域，面向对象等一系列问题。后期在设计JavaScript语言的时候，W3C就推出了一个新的版本，这个版相比5.1的版本来说扩展了很多新的功能，同时也弥补了之前的一些缺点

EMCAScript6它发布的日期是2015年，所以常ES6就是ECMAScript 2015，这个版本的JS主要对原来的版本做了如下几个扩展

1. 变量的定义
2. 解构
3. 字符串的扩展
4. 数组的扩展

5. 函数的扩展
6. 对象的扩展
7. 数据类型的扩展
8. 结构的扩展（Set与Map）
9. 反射与代理
10. `Symbol`数据类型
11. 生成器与迭代器
12. 同步与异步的扩展
13. 模块化

---

## let变量的定义

在之前的时候我们定义变量使用是关键字`var`来定义变量，这种方式定义的变量有一些问题一直存在

1. `var`定义的变量有一个声明提交（会存在一个建立阶段）
2. `var`定义的变量是没有块级作用域（只有在函数里面才有作用域）

```
console.log(a);
var a = 123;
console.log(a);
//这个时候 第1次打印的a是undefined,第二次打印的a是“hello”
```



同时，看下面代码

```
{
    var a = "hello";
}
console.log(a);
//这个时候代码执行以后打印的结果a也是"hello"
```



为了解决上面的定义变量的问题，ES6推出了新的定义变量的关键字`let`，它的定义方式就有根本的改变

```
 console.log(a);        //在这里代码会报错
 let a = 123;           //没有建立阶段，只能先定义，后调用
 console.log(a);
```



同时继续

```
{
    let a = "biaogege";
    //通过let定义的变量，它是有作用域的，以花括号为作用域
}
console.log(a);    //这个时候代码会报错
```



let定义变量的两个特点

1. `let`定义的变量你不能在定义变量之前调用它，它只是声明了这个变量存在，但是就是不能在定义之前调用
2. `let`定义的变量是有作用域的，并且是以`{}`为作用域
3. `let`定义的变量不允许在一个作用域内多次定义
4. `let`当前的作用域没有就会从外部去调用，这一点与`var`是保持一致的

## 暂时性死区

当`let`使用不当就会造成一种暂时性的死区，如下代码所示

```
let a = "biaogege";
function abc(){
    console.log(a);             //这行代码就是一个死区
    let a = "杨标老师";
    console.log(a);
}
abc();
```

softeem · 杨标

后期在使用`let`定义变量的时候一定要养成一个习惯，把所有的变量都提前定义，把在代码的最前面

## let使用的注意事项

let 在使用的时候有时候会形成一个天然的闭包关系，它解决了我们之前的闭包这种复杂的写法

**要求：创建10个定时器，每个定时器隔1秒打印1个数，从1开始，依次递增**

**错误的写法**

```
for (var i = 1; i <= 10; i++) {
    setTimeout(function () {
        console.log(i)
    }, i * 1000);
}
```

softeem · 杨标

> 代码分析：
>
> 1. 上面的代码i从1~10，创建了10个定时器，每个定时器打印i
> 2. 定时器的时间不一样，分别是1000到10000，也就是1s到10s
> 3. i在一瞬间执行完毕了，这个时候的i是用var定义的，没有区域性，所以是全局的，setTimeout里面的i就是全局的i，所以打印的结果都是11

**解决方法一**

```
for (var i = 1; i <= 10; i++) {
    setTimeout(function (j) {
        return function () {
            console.log(j);
        }
    }(i), i * 1000)
}
```

softeem · 杨标

> 代码分析：
>
> 1. 我们在`setTimeout`上面使用了闭包函数的写法
> 2. 我们返回了一个函数`function`，这里不仅是返回了一个函数，还返回了函数的执行环境
> 3. 执行环境是隔离的，所以每次隔离的变量都不一样的，这样每次隔离出来的变量就是`1,2,3....`

这种方法写起来很麻烦，所以我们在`ES6`里面专门使用`let`也可以解决这个问题

**解决方法二**

```
for (let i = 1; i <= 10; i++) {
    function a() {
        console.log(i);
    }
    setTimeout(a, i * 1000);
}
```

softeem · 杨标

> 代码分析：
>
> 1. for循环里面的`i`是使用let来定义的，它有区域性，在for循环结束以后这个`i`变量就会消失
> 2. 但是我们的定时器又是要在后面使用这个i的，怎么办呢
> 3. 所以每次在把值给到定时器里面的函数的时候，都是把i以前当前的环境直接给定时器了
> 4. 执行环境是隔离的，所以每次隔离的变量都不一样的，这样每次隔离出来的变量就是`1,2,3....`

## const定义常量

常量与变量是相对应的过程，可以变化的数据我们称之为变量，不能变化的数据我们就称之为常见

之前定义变量我们忆经学过了`var`和`let`，常量的定义则使用`const`来进行

**const具备之前let所具备的所有特性**

1. 不能在定义之前调用
2. 有作用域，以花括号形成作用域
3. 不能在一个作用域内多次定义
4. const定义的值不能改变

```
const a = "标哥哥";
a = "biaogege";              //常量是不能够改变值的，这个地方就会报错
console.log(a);

console.log(b);          //这里也会报错，不能在定义之前使用
const b = "hello world";

{
    const c = "hello world";         //这个时候的c是一个局步变量
}
console.log(c);
```

softeem · 杨标

### 锁栈与锁堆

const定义的变量不可更改这句话是有歧义的，因为const只锁栈，没有锁堆

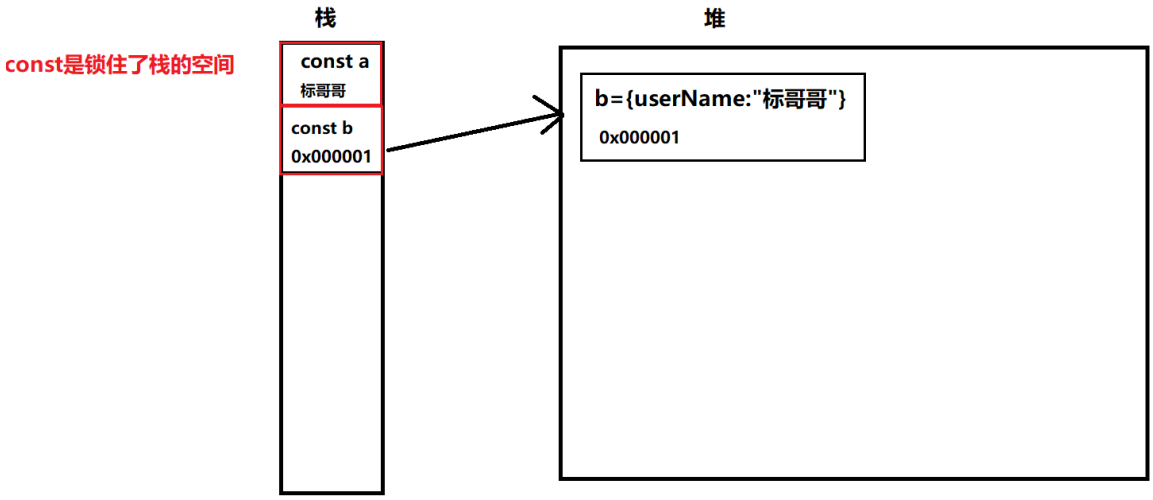
```
const a = "标哥哥";              //在栈里面定义了常量a，并锁住了这个空间
a = "邓娜";                      //当我们再次在这个空间赋值的时候，因为锁住了，所以会报错
```

softeem · 杨标

现在我们换一个例子试一下

```
const b = {
    userName:"标哥哥"
}
```

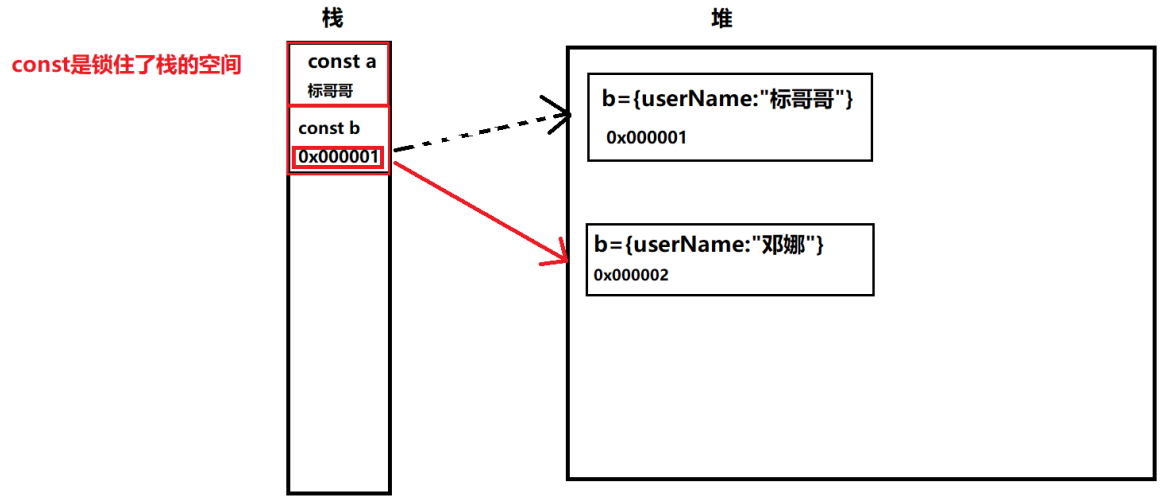
softeem · 杨标




我们现在去做下一步的更改，我们直接将一个对象赋值给了b，这个时候看一下内在的改变

```
b = {userName:"邓娜"};
```

softeem · 杨标




根据内存图来看，我们发现我们要改b在栈里面存放的地址，这栈又被 `const` 锁住了，这是不能改了

现在我们来做另一步操作

```
const a = {
    userName:"标哥哥"
}
a.userName = "邓娜";
```

softeem · 杨标

这个时候我们的代码是不会报错的，因为我们根本就没有更改a在栈里面的值，只是找到了堆里面去更改了堆里面的数据

**注意**：下面这种情况不要写，写了不会报错，但没有任何意思

```
const a;        //不赋值，不初始化，这是没有任何意思
```

softeem · 杨标

所以在使用const定义变量的时候最好就直接初始化掉

## 解构

解构是变量的一种赋值形式与取值形式，它是针对数组与对象存在的；解构只与操作方式有关，不与关键字相关。

## 解构取值

### 数组的解构取值

```
/*  ES5的语法
var arr = ["邓娜", "蒋雨晴", "易兰"];
var x = arr[0];
var y = arr[1];
var z = arr[2];
*/

let [x, y, z] = ["邓娜", "蒋雨晴", "易兰"];
console.log(x);     //邓娜
console.log(y);     //蒋雨晴
console.log(z);     //易兰
```



在解构取值的过程当，必不是需要一一对应的过程

```
let [x, y] = ["邓娜", "蒋雨晴", "易兰"];
console.log(x);
console.log(y);
```



上面的代码就相当于把邓娜赋值给`x`，把蒋雨晴赋值给`y`

```
let [x, y, z, a] = ["邓娜", "蒋雨晴", "易兰"];
console.log(x);
console.log(y);
console.log(z);
console.log(a);          //它没有解构出值来，所以是undefined
```



### 对象的解构取值

```
let stu = {
    stuName: "邓娜",
    stuSex: "女",
    stuAge: 18,
    stuAddress:"湖北武汉"
}
/* ES5的写法
let stuName = stu.stuName;
let stuSex = stu.stuSex;
let stuAge = stu.stuAge;
*/

let { stuName, stuSex, stuAge } = stu;
console.log(stuName);
console.log(stuSex);
console.log(stuAge);
```



对象在解构取值的时候是通过属性来进行解构的，如果有这个对象有这个属性名，就拿出这个属性名的值，如果没有就是`undefined`

当我们的对象与数组比较复杂的时候，我们可以通过下面这种方式来参执行

```
let obj = {
```

```
    userName: "标哥哥",
    sex: "男",
    hobby: ["看书", "睡觉"],
    compute: {
        comName: "小米",
        price: 3000
    }
}
// 解构取出如下值，姓名，第1个爱好，电脑的价格与品牌

let { userName, hobby: [hobby1], compute: { comName, price } } = obj;
console.log(userName);
console.log(hobby1);
console.log(comName);
console.log(price);
```

### 解构赋值

解构的赋值发生在对象里面，先看下面的代码

```
let userName = "标哥哥";
let sex = "男";

let stu = {
    userName: userName,
    sex: sex
}
console.log(stu);
```

softeem · 杨标

在上面的代码过程当中，当我们把对象进行封装的时候，我们发现了一点，**它的属性名与属性值是一样的**，这个时候，我们可以进行解构赋值

```
let userName = "标哥哥";
let sex = "男";
let obj = {
    userName,sex
}
console.log(obj);
```

softeem · 杨标

这个时候我们发现结果与上面是一模一样的

**小技巧**：我们还可以通过解构快速的将变量的值进行互换

```
let a = 10;
let b = 20;

//有没有什么办法将上的两个值互换
[b, a] = [a, b];
console.log(a,b);
```

softeem · 杨标

## 展开运算符

展开运算符是`ES6`当中新出的一种运算符，它使用`...`来表示，可以把一个具备`Iterable`接口的数据展开。目前为上有以下几个数据类型是具备这个`Iterable`接口的

> 只要是具备`Iterable`接口的就一定会有一个方法叫`Symbom.Iterator()`

1. `Array`数组
2. `arguments`
3. `NodeList`与`HTMLCollection`
4. `TypeArray`类型数组
5. `Set`单值集合
6. `Map`键值对集合

**第一个例子**

```
let arr = ["张三", "李四", "王五", "赵六"];
// console.log(arr[0], arr[1], arr[2], arr[3]);
console.log(...arr);            //展开了这个arr
```



**第二个例子**

```
let nums = [1, 4, 6, 9, 2, 7, 9, 11, 3];
//求出上面数组中的最大值
/* 第一步：直接写
let max = Math.max(1, 4, 6, 9, 2, 7, 9, 11, 3);
console.log(max);
*/
/*第二步：通过apply传参数的特点
let max = Math.max.apply(Math, nums);
console.log(max);
*/

let max = Math.max(...nums);
console.log(max);
```



**第三个例子**

```
let arr1 = ["邓娜", "蒋雨晴"];
let arr2 = ["易兰", "徐大江"];
//将arr2的数据放到arr1当中去，怎么办

/* 第一种思路 ：使用concat
arr1 = arr1.concat(arr2);
console.log(arr1);
*/
/*第二种思路：使用push
arr1.push(arr2[0], arr2[1]);
console.log(arr1);
*/

arr1.push(...arr2);
console.log(arr1);
```



**注意**：展开运算符并不是所有的元素都可以展开，只有具备`Iterable`这个接口的才可以展开

```
let obj = {
    userName: "标哥哥",
    sex: "男",
    age: 18
}

console.log(...obj);        //代码在这里会报错
```

自定义的对象上面是没有实现`Iterable`这个接口的，所以不能进行展开

## 展开运算符的应用

### 场景一：数组的展开应用

```
/**
 * 展开运算符
 * [1, 2, 3].map(function (item) { return "./assets/enemy0_down" + item + ".png" }) 它必然是一个数组
 * 我们可以认为它是 ["./assets/enemy0_down1.png","./assets/enemy0_down2.png","./assets/enemy0_down3.png"]
 */
/*
let res = [
    "./assets/bomb.png"
].concat([1, 2, 3].map(function (item) { return "./assets/enemy0_down" + item + ".png" }));
*/

/*
let res = [
    "./assets/bomb.png",
    ...[1, 2, 3].map(function (item) { return "./assets/enemy0_down" + item + ".png" })
]
*/

let tempArr = ["张三", "李四", "王五"];
let arr = [
    "a", "b", "c", ...tempArr
]
console.log(arr);
```

> 在上面的场景 里，我们使用了展开运算符将数组里面的元素展开，放到了一个新的数组里面

### 场景二：数组的简单拷贝

```
let arr1 = ["a", "b", "c", "d", "e"];
/*
let arr2 = arr1;        //做了一次浅拷贝
arr2.push("f");
console.log(arr1);
*/
//如果想快速的对这种数组做深拷贝 ，怎么办呢
/*
let arr2 = arr1.concat();
```

```
//或
let arr3 = arr1.slice();
arr2.push("张三");
arr3.push("李四");
console.log(arr1);
*/
//现在新增了一种方式

let arr2 = [...arr1];
arr2.push("新元素");
console.log(arr1);
console.log(arr2);
```

**场景三：关于对象展开以后，与解构赋值同时使用**



```
let obj = {
    userName: "邓娜",
    age: 18
}
//如果想得到一个深拷贝的对象，怎么办
// let obj1 = obj; //浅拷贝

//思路一：取值以后再赋值
/*
//解构取值
let { userName, age } = obj;
// 解构赋值
let obj2 = {
    userName, age
}
obj2.userName = "易兰";
console.log(obj.userName);
console.log(obj2.userName);
*/
//思路二：我们之前说过对象是不允许展开的，但是下面的方式是允许的
let obj2 = { ...obj };
obj2.userName = "易兰";
console.log(obj);
console.log(obj2);
```

**场景四：对象的扩展与拷贝**



```
let obj = {
    userName: "邓娜",
    sex: "女"
}
//我现在有第2个学生，这个学生就是obj对象上面的扩展，它多了1个新的属性叫age:18,怎么办呢
//并且两个对象之前互不影响
/*第一种思路：最原始的思路
let obj2 = {
    userName: "邓娜",
    sex: "女",
    age: 18
}
*/
/**
```

```
 * 第二种思路：ES5里面的Object.assin()这个方法
 */
/*
let obj2 = {};
Object.assign(obj2, obj, { age: 18 });
obj2.userName = "易兰";
console.log(obj2,obj);
*/
/**
 * 第三种思路：ES6里面的展开运算符+解构赋值
 */

let obj2 = { ...obj, age: 18 };
obj2.userName = "徐大江";
console.log(obj2, obj);
```

# 字符串扩展

在ES6里面，字符串的扩展主要有两个方向，一个是扩展了模板字符串，另一个则是扩展了字符串中的一些方法

## 模板字符串

模板字符串是为了更好的控制字符串的格式与内容而产生的一个技术，它可以快速的实现字符串排版，拼接，替换等一系列内容

模板字符串的定义也非常简单，它使用反引号来完成



```
let userName = "标哥";
let age = 18;
let str = "大家好，我叫" + userName + ",我今年" + age + "岁了";
//所有后面的时候我写了一个字符串的正则表达式的扩展方法 叫"format"

let str1 = `大家好，我叫${userName}，我今年${age}岁了`;
console.log(str);
console.log(str1);
```

模板字符串可以让我们快速的实现字符串的排版与字符串的拼接过程



```
/**
 * 模板字符串的排版
 */

function sendMail(userName, userId, registerTime) {
    let str = `
    尊敬的${userName}:
    您好!
    感谢你注册我们商城的账号，您的账号是${userId}，您的注册时间是${registerTime}!
    如果有任何疑问，请联系客服。
    客服QQ：1234567
    客服电话：12456789

                                        标哥商城
                                        ${new Date().toLocaleString()}
 `;
    console.log(str);
```

```
}

sendMail("标哥哥", "yangbiao", "2021-1-15");
```

## 字符串模板的注意

### 注意点一：模板字符串与方法的结合

```
let userName = "标哥哥";
let age = 18;
function abc() {
    console.log(arguments);
}

abc`大家好，我叫${userName},我的年龄是${age}岁了`;
```



上面的代码打印以后结果如下

```
[Arguments] { '0': [ '大家好，我叫', ',我的年龄是', '岁了' ], '1': '标哥哥', '2': 18 }
```



### 注意点二

```
let age = 18;
// let str = `大家好，我叫${userName},我今年${age}岁了`;
//上面的代码会报错，因为userName需要定义
let str = `大家好，我叫标哥哥，我今年${age + 1}岁了，我${age >= 18 ? '成年' : '未成年'}了`;
console.log(str);
```



## 字符串方法的扩展

> 因为之前在ES5里面，我没有复习字符串的方法，所以在这里我把ES5与ES6的方法全部展示出来

1. `length`属性，代表字符串的长度
2. `indexOf()`方法，在字符串中查询某个元素所在的索引位置
3. `lastIndexOf()`方法，从后向前找，找索引，找不到就是-1
4. `replace()`方法，替换字符串
5. `slice(start,end)`方法，截取字符串，注意，这个方法与数组保持一致，start与end都可以取负值
6. `substring(start,end)`方法，截取字符串，但它不使用负值
7. `substr(start,legnth)`方法，截取字符串，但第2个参数是长度
8. `toUpperCase()`方法，转换成大写字终
9. `toLowerCase()`方法，转换成小写字终
10. `search()`方法，查询符合要求字符的位置，它与`inexOf()`很像，但是`search`是支持正则表达式的
11. `split()`方法，将字符串按照符合要求的格式是拆分成数组
12. `trim()`方法，去除字符串左右的空格
13. `trimLeft()/trimRight()`方法，去除左边或右边的空格

14. `trimStart()/trimEnd()`方法，去除开始与结束的空格【ES6字符串中新增的方法】
15. `padStart/padEnd()`方法，在当前字符串的开始或结束的位置使用指字的定符填充指定的长度【ES6字符串中新增的方法】
16. `includes()`方法，当前字符串中是否包含了某个字符【ES6字符串中新增的方法】
17. `match()`方法，根据正则表达式匹配符合要求的内容
18. `matchAll()`方法，根据正则表达式匹配符合要求的内容，它的正则表达式必须的修饰符`g`

```
var str = "我的出生日期是1999-12-2，我女朋友的生日是2000-3-15,我弟弟的生日是2005-5-7";
//我现在想提取所有的日期
//这一种方式，我们称之为简单的匹配
var result1 = str.match(/\d{4}-\d{1,2}-\d{1,2}/g);
//第二种情况，我希望把所有年放在一个数组，月放在一个数组，日放在一个数组
var year = [];
var month = [];
var day = [];

var reg = /(\d{4})-(\d{1,2})-(\d{1,2})/g;

var result2 = str.matchAll(reg); //它会返回一个可以遍历的对象，然后将每次正则的结果告诉你
//在这个结果当中，你可以获取到原子组
var temp = null;
while ((temp = result2.next()).done == false) {
    year.push(temp.value[1]);
    month.push(temp.value[2]);
    day.push(temp.value[3]);
}
```



`matchAll`相较于`match`来说最大 的优点就是不光可以匹配正则内容，还可以拿到原子组的内容

19. `startsWith()/endsWith()`判断字符串是否以什么开头或什么结尾
20. `concat()`方法，与数组里面的方法保持一致，拼接一个字符串，形成一个新的字符串，原字符串不变
21. `charCodeAt()`方法，获取当前字符串的`unicode`码，如果是英文或号或数字则是`ascii`码
22. `String.fromCharCode()`方法，从某一个`ascii`码得到一个字符

## 数组的扩展

数组在ES6里面扩展的时候，主要扩展的就是方法

1. `Array.isArray()`方法

   该方法判断某一个变量是否是否是一个数组

2. `Array.of()`方法

   这个方法主要是为了解决数组在定义的时候存在的一些歧义

```
let arr1 = new Array(6);
//上面的这个6代表的是数组的长度
```



如果我们希望在定义数组的时候，里面默认就有一个元素是6，怎么办呢

```
let arr2 = [6];
//或
let arr3 = Array.of(6);      //这个时候的6代表的就是里面的元素
```

同时它还可以放多个元素

```
let arr4 = Array.of(5, 6, 7, 8, 9);
```

3. `Array.from()`将一个类数组转换成数组

在ES5里面，我们如果要将一个类数组转换成数组，推断过程相当麻烦

```
let obj = {
    0: "张三",
    1: "李四",
    2: "王五",
    length: 3
}
//将上面的对象转换成数组，怎么办呢？
//Array数组下面有个slice方法，它是截取数组的元素形成一个新的数组，它的返回值一定是一个数组
let arr = [1, 2, 3, 4, 5];
let arr1 = arr.slice();  //这个时候的arr1必然是一个新的数组
//思路：我们能不能让obj也去调用arr里面的slice方法呢
//第一步：
arr.slice == arr.__proto__.slice;
//第二步：如果直接调arr.__proro__.slice()方法，这个时候this指向的是arr的__proto__,所以我们要让this偏移
arr.__proto__.slice.call(arr);
//第三步：我们又知道 一个对象的__proto__等于它的构造函数的prototype
arr.__proto__ == Array.prototype;
Array.prototype.slice.call(arr);
//第四步：让this指向obj
var result1 = Array.prototype.slice.call(obj);

console.log(result1);
```

**现在在ES6里面，要把类数组转成数组非常简单了**

```
let obj = {
    0: "张三",
    1: "李四",
    2: "王五",
    length: 3
}
let result2 = Array.from(obj);
console.log(result2);
```

4. `Array.prototype.fill()`方法

该方法是使用指定的值填充数组

```
let arr = new Array(10);
//数组的长度是10
arr.fill("标哥");
```

5. `Array.prototype.flat(steps)`方法，拍平数组

   这个方法是把多维数组进行拍平，里面的参数`steps`代表你是拍平几层，如果想全部拍平，则参数设置为`Infinity`

```
let arr = [1, [2, 3,[4,5,[6,7,[8,9,[10,11]]]]]];
arr.flat(2);    //拍平2次
arr.flat(3);    //拍平3次
arr.flat(Infinity);     //传入的参数Infinity是无穷大，数组会全部拍平
```



6. `Array.prototype.findIndex()`方法，根据要求查找指定的元素的索引位置

   这个方法看起来与我们之前所学习的`indexOf()`非常相似，但是它比`indexOf`更高级一点

   找到以后返回索引，找不到返回的是`-1`

```
let arr = ["a", "b", "c", "d", "e", "f"];
//如果想找d在什么位置，怎么办?
let index1 = arr.indexOf("d");

let index2 = arr.findIndex(function (item, index, _arr) {
    return item == "d";
});

console.log(index1,index2);
```



   通过上面的例子，我们感觉`indexOf`比下面的`findIndex`更简单

   但是如果碰到复杂的情况，则有问题

```
let arr = [
    { uid: 1, userName: "张三", age: 19 },
    { uid: 2, userName: "李四", age: 19 },
    { uid: 3, userName: "王五", age: 19 },
    { uid: 4, userName: "邓娜", age: 19 },
    { uid: 5, userName: "蒋雨晴", age: 19 }
]
//我想找邓娜在哪个位置，怎么办?
//这个时候indexOf就派不上用声了
//第一种思路，原始思路
let index = -1;
for (let i = 0; i < arr.length; i++) {
    if (arr[i].userName == "邓娜") {
        index = i;
        break;

    }
}
```



   **现在使用`findIndex`来完成**

```
let index = arr.findIndex(function (item, index, _arr) {
    return item.userName == "邓娜";
})
console.log(index);
```



7. `Array.prototype.find()`根据要求查找元素

这个方法可以与上面的`findIndex`做对比，也可以与之前所学习的`filter`去做一个对比

`find`方法如果找不到符合要求的元素，最终的结果就是`undefined`

```
let arr = [
    { uid: 1, userName: "张三", age: 19 },
    { uid: 2, userName: "李四", age: 19 },
    { uid: 3, userName: "王五", age: 19 },
    { uid: 4, userName: "邓娜", age: 19 },
    { uid: 4, userName: "邓娜", age: 20 },
    { uid: 5, userName: "蒋雨晴", age: 19 }
]
//我现在希望拿到userName为邓娜，age为20的怎么办？
//第一种思路：以前的方法
let arr1 = arr.filter(function (item, index, _arr) {
    return item.userName == "邓娜" && item.age == 20;
});
console.log(arr1);
//上面的方法得到的是数组，我们还从数组里面拿到值
//第二中思路：使用find完成
let result = arr.find(function (item, index, _arr) {
    return item.userName == "邓娜" && item.age == 20;
});
console.log(result);
```



## for...of的遍历

之前的时候我们在遍历数组的时候已经有很多种方法了，如下所示

```

//之前在学习数组的时候，关于数组的遍布已经不是一天二天了
//我们也掌握了很多方法去遍历数组
let arr = ["a", "b", "c", "d", "e"];
//第一种方式：  for
for (let i = 0; i < arr.length; i++) {
    // i代表索引
    console.log(arr[i]);
}
//第二种方式：for...in
for (let i in arr) {
    //i代表索引
    console.log(arr[i]);
}
//第三种方式的遍历
arr.forEach(function (item, index, _arr) {
    // index代表索引
    console.log(item);
});
```



在上面的三种遍历方式当中，我们每次遍历的结果都会有索引。**也就是说这三种方式都是依赖于索引去遍历的**

想象一下，如果一个数据集合里面没有索引了，我们怎么遍历？

这个时候我们要考试到一点，如果当前集合没有索引了，我们仍然要遍历，怎么办呢？在ES6里面就推出一个新的试叫`for...of`，它不依赖于索引遍历。也就是说这个集合即使没有索引，我仍然可以遍历

# for...of的遍历只通过`Iterable`来实现的

```
let arr = ["a", "b", "c", "d", "e"];
//大家都知道，数组是具备Iterable的接口的，它内部有一个Symbol.Iterator这个方法
//有了这么个东西，它就可以使用for...of来遍历
for (let item of arr) {
    console.log(item);
}
```



## Set单值集合

它是ES6里面出的一个数组结果，同学们可以把它看成是一个没有索引(key)的不重复的数组

### `Set`的创建

Set 是一个构造函数，需要new出来，所以它的创建方式如下

```
let s1 = new Set();
console.log(s1);            //Set {}

let s2 = new Set(["a", "b", "c", "d"]);
console.log(s2);            //Set {"a","b","c","d"}
```



通过上面的东西我们可以看到，我们分别创建了一个空的Set和有内容的Set，Set里面存放的元素是没有索引的

同时我们要注意另外一个特性，Set是不重复的集合，如果重复了，会自动过滤掉重复的元素

```
let s1 = new Set(["a", "b", "c", "d", "b", "d"])
console.log(s1);            //Set  {"a","b","c","d"}
```



我们在上面的代码里面已经看到，它把重复的元素过滤掉了

🏴 **小技巧**：我们可以根据这个特点来实现数组的去重

```
let arr = ["a", "b", "c", "a", "a", "b", "d", "e", "f", "c", "g", "d", "e",
"g", "a", "c", "a", "h", "a"];
// 上面是一个重复的数组
let s1 = new Set(arr);
console.log(s1);
//第二步：只需要将上面的s1转换成数组即可
//第一种方式：Array.from()  它接收Iterable这个接口的参数，我们的Set有没有实现这个接口呢？
//通过观察发现Set实现了Iterable这个接口，具备Symbol.Itertor这个方法
let arr1 = Array.from(s1);
console.log(arr1);
//第二种方式：我们已经知道Set实现了Iterable这个接口，所以我们还可以
let arr2 = [...s1];
console.log(arr2);
```



### `Set`的相关方法

Set做为一个新的结构类型，它应该与数组一样，有一些操作自己的方法

1. `size`属性，获取当前这个集合里面的元素数量
2. `add()`向集合中新增一个元素
3. `delete()`从集合中删除一个元素
4. `has()`判断某一个元素是否在集合里面
5. `clear()`清空当前集合
6. `values()`获取当前Set容器里面的值的集合，返回一个**迭代器**
7. `keys()`获取当前Set容器里面的键的集合（因为它没有键，所以它的结果也是values的结果），返回一个**迭代器**

### Set思考

1. Set是单值集合，它没有索引（没有key），那么的它的排序是怎么样的？

   它因为没有索引，所以在内在当中去存储数据的时候是可以不用连续放在同一块内存区域的，所以它的序号我们控制不了，最终得出的结论就是Set是一个单值的无序集合

2. Set是单值集合，没有索引，那么，它能不能遍历呢？怎么遍历呢？ 首先我们如果要遍历Set集合，那么必须找到一个不依赖于索引来遍历的方法，这个时候只有`for...of`

   使用`for...of`的前提条件这个对象必须具备`Iterable`这个接口，而`Set`是具备这个接口的，所以我们可以使用`for...of`来进行遍历

### Set的遍历

```
let s1 = new Set(["a", "b"]);
s1.add("邓娜").add("蒋雨晴");
for (let item of s1) {
    console.log(item);
}
```



## Map键值对集合

> 在讲这个东西之前，先说明一个概念，我们把键可以理解为属性名，把值可以理解为属性值

### Map的创建

Map的创建其实与对象都差不多，因为我们在之前学习对象的时候，我们也说过对象其实也是一个键值对

```
let m1 = new Map();     //这里创建的是一个空的map集合
m1.set("01", "邓娜");
m1.set("02", "蒋雨晴")
```



当然还可以直接给定初始值

```
let arr = [
    ['a', '标哥'],
    ['b', '张三'],
    ['c', '李四']
]
let m1 = new Map(arr);
```



### Map的应用场景

```
let stuList = [
    { sid: "H20030001", stuName: "张三", sex: "女", age: 19 },
```

```
    { sid: "H20030002", stuName: "李四", sex: "女", age: 19 },
    { sid: "H20030003", stuName: "王五", sex: "女", age: 19 },
    { sid: "H20030004", stuName: "赵六", sex: "女", age: 19 },
    { sid: "H20030005", stuName: "曾七", sex: "女", age: 19 }
];
//在上面的数组里面，我想快速的根据某个学号找到某个学生，怎么办？
function getStuById(id) {
    let result = stuList.find(function (item) {
        return item.sid == id;
    });
    return result;
}
```

上面的方法是我们自己写了一个方法，然后通过`find`去找。现在我们换一个思路



```
let m1 = new Map();
m1.set("H20030001", { sid: "H20030001", stuName: "张三", sex: "女", age: 19 });
m1.set("H20030002", { sid: "H20030002", stuName: "李四", sex: "女", age: 19 });
m1.set("H20030003", { sid: "H20030003", stuName: "王五", sex: "女", age: 19 });
m1.set("H20030004", { sid: "H20030004", stuName: "赵六", sex: "女", age: 19 });
m1.set("H20030005", { sid: "H20030005", stuName: "曾七", sex: "女", age: 19 });

let result = m1.get("H20030001");
console.log(result);
```

> 通过一个键快速的去找到某个值，这种做法是最方便的

## Map的常用方法

1. `size`属性，获取当前集合元素的个数
2. `set(k,v)`向集合里面添加一个新的键值对
3. `get(k)`根据一个键向集合当中取值，如果取不到，就是undefined
4. `delete(k)`根据一个key去删除一条记录
5. `clear()`清空当前的键值对
6. `keys()`获取当前Map容器里面的键的集合，返回一个**迭代器**
7. `values()`获取当前Set容器里面的值的集合，返回一个**迭代器**

## Map的遍历

首无map也是一个无序的集合，只是说它有键了，便是它没有索引，它的遍历方式还是只能够使用`for...of`来进行

```
let m1 = new Map();
m1.set("H20030001", { sid: "H20030001", stuName: "张三", sex: "女", age: 19 });
m1.set("H20030002", { sid: "H20030002", stuName: "李四", sex: "女", age: 19 });
m1.set("H20030003", { sid: "H20030003", stuName: "王五", sex: "女", age: 19 });
m1.set("H20030004", { sid: "H20030004", stuName: "赵六", sex: "女", age: 19 });
m1.set("H20030005", { sid: "H20030005", stuName: "曾七", sex: "女", age: 19 });
for (let item of m1) {
    console.log(item);   //键值对里面的某一项
}
```



上面我们直接使用`for...of`去遍历的时候，得到的每一项都是一个键值对，它使用下面的形式表示

```
[ 'H20030001', { sid: 'H20030001', stuName: '张三', sex: '女', age: 19 } ]
```



如果我们想快速的得到键与值，这个时候我们可以使用解构来完成

```
for(let [k,v] of m1){
    //这个时候解构出来的k代表的就是键，解构出来的v代表的就是值
}
```



**注意事项**：它里的map虽然已经有键了，但是我们仍然不能使用`for...in`去遍历。原因在于`for...in`只适合在内在当中连续存储的数据来遍历，而map与set是一样的，都是一个散列存储，在内在当中是一个无序结构

同时注意一点，map的键`key`是不允许重复的，如果重复了，则是后面的覆盖前面的；而值`value`则可以重复

---

## 函数的扩展

函数一直都是ES最基础的东西，它是实现低耦合最关键的东西，它也是实现模块的基本，更是封装的底层，还是面向对象开发的一个本质，更是执行上下文的一种体现

在ES6里面，对函数又作了进一步的扩展，主要就体现在下面

### 无构造函数的函数

```
let obj = {
    age: 18,
    Student: function () {
        this.userName = "标哥";
    }
}
let s1 = new obj.Student();
console.log(s1);
```



上面的写法一点问题都没有，因为`Student()`这个函数在对象里面，但是它仍然是一个函数，在ES5里面，只要是函数，我必然可以`new`来调用

但是在ES6里面，则要注意。上面的代码在正常情况下，应该是下面的方式

```
let stu = {
     userName:"邓娜",
     age:18,
     sayHello:function(){
         console.log(`大家好，我叫${this.userName},我的年龄是${this.age}`);
     }
 }
```



在上面的对象里面`sayHello`做为`stu`对象的一部分，它存在的目的不应该是以构造函数的形式存在的，而应该只是对象下面的一个普通方法，但是在ES5里面，仍然是可以通过`new`来调用的，这种场景是严重不合理

```
let obj = new stu.sayHello();
 console.log(obj);
//上面的代码不会错，但是并不合理
```



为了解决这个问题，在ES6里面就出了一个新的函数定义方式，如下

```
let stu2 = {
    userName: "邓娜",
    age: 18,
    sayHello() {
        //这就是一个无构造函数的函数
        console.log(`大家好，我叫${this.userName},我的年龄是${this.age}`);
    }
}

stu2.sayHello();
let obj2 = new stu2.sayHello();          //直接报错，因为sayHello没有构造函数，不能new
```



这个时候我们把`stu2.sayHello`这个方法的定义形式改变了下，这个`stu2.sayHello`这个方法就不具备`constructor`构造函数的这个特性了，所以它不能再`new`调用了

## 函数参数的默认值

这前在定义函数的时候，函数是可以定义的参数

```
function sayHello(userName){
    console.log(`大家好，我叫${userName}`);
}

sayHello("标哥哥");
sayHello();
```



上面的参数在不给值`userName`的时候它就是`undefined`,那么有没有一个办法在我不设置值的时候就给一个默认值呢?

**ES5的思路**

```
function sayHello(userName) {
    userName = userName || "玉皇大帝";
    console.log(`大家好，我叫${userName}`);
}

sayHello("标哥哥");
sayHello(0);
```



以前的时候我们是使用关系运算符来完成，但是这么做有一个缺点，如果我们的参数是`0`，这样它也会得一个假的条件，最后再取默认值，这是不合理的

**ES6的思路**

在ES6的语法里面，允许在定义参数的时候给一个默认值

```
function sayHello(userName = "玉皇大帝") {
    console.log(`大家好，我叫${userName}`);
}

sayHello("标哥哥");
sayHello();
sayHello(0);
```



这个时候我们就在定义参数`userName`的时候给了一个默认值，当如果对这个参数不赋值，则使用默认值来进行

**注意事项**

```
function sayHello(userName = "王母娘娘",age){
    console.log(`大家好，我叫${userName},我的年龄是${age}`);
}

sayHello("标哥哥",19);
//如果我现在想使用userName的默认值，age为10000岁，是否可行？
sayHello(10000);
```



上面的写法是错误的，函数的参数如果有了默认值，则在定义的时候只能放在后面

```
function sayHello(age, userName = "孙悟空") {
    console.log(`大家好，我叫${userName},我的年龄是${age}`);
}

sayHello(18,"标哥哥");
sayHello(10000);

function abc(age, userName = "七仙女", sex = "女") {
    console.log(`我们是${userName},我们的性别是${sex},我们的年龄是${age}`);
}

abc(50);
abc(60,"玉兔");
abc(70,"太白金星","男");
```



## 箭头函数

下面就是箭头函数的演变过程

```
/* 第一步：
function sayHello(){
   console.log("大家好，我是函数");
}
*/
/*第二步：
const sayHello = function () {
    console.log("大家好，我是函数");
}
*/
const sayHello = () => {
    console.log("大家好，我是箭头函数");
}
sayHello();
console.log(typeof sayHello);
```



**有参数的箭头函数**

```
/*多个参数，我们打括号包起来*/
const add = (a, b) => {
    let result = a + b;
    console.log(result);
}

/*一个参数则可以省略括号*/
const abc = userName => {
    console.log("大家好，我叫" + userName);
}
```



> 0个参数与多个参数，我们会使用括号，而1个参数则括号就可以省略

现在我们已经了解了箭头函数，我们进一步简化箭头函数

```
const a = userName => {
    console.log(`大家好，我叫${userName}`);
}

const b = userName => console.log(`大家好，我叫${userName}`);
```



上面的箭头函数`a`与`b`看起来是一样的，但是效果完全不同，这个区别我们可以通过下面的代码来看

```
let add1 = (a, b) => { return a + b; }

let add2 = (a, b) => a + b;
```



箭头函数如果没有加花括号，则代表后面的表达式就是返回值。所以上面的`add1`与`add2`是相同的

同时箭头函数的演变过程也可以看下面的代码

```
let arr = [1, 2, 3, 4, 5, 6, 7, 8, 9];
//获取数组中的偶数

//第一次演变
let result1 = arr.filter(function (item, index, _arr) {
    return item % 2 == 0;
```

```
});
//第二次演变
let result2 = arr.filter((item, index, _arr) => {
    return item % 2 == 0;
});
//第三次演变
let result3 = arr.filter((item, index, _arr) => item % 2 == 0);
//第四次演变
let result4 = arr.filter(item => item % 2 == 0);
console.log(result4);
```

## 箭头函数的注意事项

1. 箭头函数的定义使用的是变量这种方式在定义，它没有声明提前，所以必须是先定义，再使用

```
add(1,2);       //报错
const add = (a, b) => a + b;
```



2. 箭头函数是没有构造函数的，不能`new`调用

```
const add = (a, b) => a + b;
let obj = new add(1,2);     //报错，不能用new调用
```



3. 箭头函数里面没有`arguments`实参集合

```
const abc = () => {
    console.log(arguments);     //这个时候这里并不是实参集合
}
abc(1,2);
```



4. 箭头函数的`this`会执行绑定过程

   通俗的说法就是箭头函数的`this`会跳过当前环境，去拿上一级环境

```
let obj = {
    userName: "标哥哥",
    sayHello() {
        console.log(this.userName);
        /*
            setTimeout(function () {
                console.log(this);
                console.log("时间到! " + this.userName);
            }.bind(this), 2000);
            */
        setTimeout(() => {
            console.log("时间到" + this.userName);
        }, 2000);
    }
}
```



正是因为有一个这样的现象，我们的回调函数一般都会使用箭头函数，**也正是因为有这个现象，所以 在对象里面定义方法的时候是不能使用箭头函数的**

```
 let obj2 = {
     userName: "标哥哥",
```

```
    sayHello() {
        console.log(this)
        console.log(this.userName);
    },
    sayHello1: () => {
        //这么定会跳过当前对象
        console.log(this)
        console.log(this.userName);
    }
}

obj2.sayHello();
obj2.sayHello1();
```

有下面一个箭头函数的案例

softeem · 杨标

```
var userName = "外边的标哥哥";
//var定义的变量在window下面
// this.userName相当于window.userName
//let就不是的
let obj = {
    userName: "标哥哥",
    sayHello: () => {
        setTimeout(() => {
            console.log(this.userName);
        }, 1000);
    }
}
```

在上面的代码当中，我们使用了两次箭头函数，这个时候最终打印的this.userName就会是最外边的`userName`了

**同时也要注意，在DOM的事件绑定里面也不要使用箭头函数**

softeem · 杨标

```
<body>
    <button type="button" id="btn1">按钮1</button>
</body>
<script>
    let btn1 = document.querySelector("#btn1");
    /*
    btn1.onclick = function(){
        console.log(this);          //指向button
    }
    */

    btn1.onclick = () => {
        console.log(this);          //指向window
    }
</script>
```

## 函数柯里化

在计算机科学中，柯里化（Currying）是把接受多个参数的函数变换成接受一个单一参数(最初函数的第一个参数)的函数，并且返回接受余下的参数且返回结果的新函数的技术。

### 优点一

```
function sayHello(userName = "标哥", sex = "男", age = 18) {
    console.log(`大家好，我叫${userName},我的性别${sex},我的年龄${age}`);
}
sayHello("张杨明","男",20);
```



在上面的函数当中，如果我们想把第2个参数使用默认值，这么做是不可以的

```
function sayHello(userName = "标哥") {
    return function (sex = "男") {
        return function (age = 18) {
            console.log(`大家好，我叫${userName},我的性别${sex},我的年龄${age}`);
        }
    }
}
sayHello("张杨明")()(20);
```



**分步进行**

```
/*
function checkLogin(userName, pwd) {
   //检测登陆的过程当中
   //1.用户名是否存在
   //2.密码是否正确
}
*/
//但是上面的方式如果要分步骤进行
function checkLogin(userName) {
    let list = ["admin", "biaogege"];
    //1.检测用户名是否存在
    //如果存在返回function.不存在返回 false
    if (list.includes(userName)) {
        //用户名存在
        return function (pwd) {
            //再检测用户名密码
            console.log(`你的用户名是${userName},你的密码是${pwd}`);
        }
    }
    else {
        return false;
    }
}
let result = checkLogin("admin");
if (result) {
    //说明
    result("123456");
}
else {
    //又说明什么
    console.log("用户名不存在")
}
```



## class关键字

在ES6里面`class`关键字的作用是用于创建构造函数，在之前的ES5里面，我们使用`function`来创建构造函数，但是这种写法总是会给一种有歧义的感觉，所以在ES6里面它出了一个新的关键字叫`class`，专门用于声明对象的构造函数

**ES5创建对象**

```
function Student() {
    this.stuName = "标哥哥";
    this.sayHello = function () {
        console.log(`大家好，  我叫${this.stuName}`);
    }
}

let stu1 = new Student();       //得到了一个学生对象
```



**ES6创建对象**

```
class Student {
    stuName = "标哥哥";
    sayHello() {
        console.log(`大家好，  我叫${this.stuName}`);
    }
}

let stu1 = new Student();
stu1.sayHello();
```



在上面的代码当中，我们分别使用`ES5`与`ES6`两种方式去创建了对象，并且都成功了

同时我们在通过`typeof Student`检测的时候，它们的结果都是`function`类型，那就说明下面的`class Student`创建的还是一个`function`，所以我们可以认为`ES6`里面的`class Student`与`ES5`里面的`function Student`所达到的效果是相同的

如果在创建构造函数的时候有了参数，则应该向下面这种方式处理

**ES5的代码**

```
function Student(stuName) {
   this.stuName = stuName;
   this.sayHello = function () {
       console.log(`大家好，  我叫${this.stuName}`);
   }
}
let stu1 = new Student("邓娜");
```



**ES6的代码**

```
class Student {
    //在每个class的内部，有一个函数是默认存在的,叫constructor
    //它会在new的一瞬间  自动执行
    //constructor是构造函数，构造函数里面的this指向了当前对象
    constructor(stuName){
        this.stuName = stuName;
    }
    sayHello(){
        console.log(`大家好，  我叫${this.stuName}`);
```

```
    }
}

let stu1 = new Student("邓娜");
stu1.sayHello();
```

## class里面的`static`关键字

`static`的意思是静态的意思，它本意是指内在里面的一块静态空间，但是在ES5里面则指的是另一层函数。它是一个修饰符，用于修饰属性或方法，它有一个特点就是不需要`new`出对象就可以直接使用【一这点是模拟了java与其它编程语言的操作】



```
class Student {
    constructor(stuName) {
        this.stuName = stuName;
    }
    sayHello() {
        console.log(`大家好，我叫${this.stuName}`);
    }
    //静态的方法
    static abc() {
        console.log("大家好，  我是abc这个方法");
    }
    //静态的属性
    static age = 18;
}

let stu1 = new Student("邓娜");
stu1.sayHello();
//这个时候的abc方法使用了关键字static去修饰，我们再调用这个方法的时候不需要使用new的对象去调用，直接使用class创建构造函数调用
Student.abc();
console.log(Student.age);
```

在我们使用class创建对象的构造函数的时候，可以设置静态与非静态，那么它们之间是否可以互相调用



```
class Student {
    constructor(stuName) {
        this.stuName = stuName;
    }
    sayHello() {
        console.log(`大家好，我叫${this.stuName}`);
        console.log(`我的年龄是${this.age}`);            //undefined
        console.log(`我非要调用你的年龄${Student.age}`);    //18
    }
    //静态的方法
    static abc() {
        console.log("大家好，  我是abc这个方法");
        console.log(`静态方法调静态变量${Student.age}`);
        console.log(`我在静态里面调stuName，${this.stuName}`);  //undefined这是绝对不允许的
    }
    //静态的属性
    static age = 18;
}
```

**总结**：非静态的方法与变量都是需要`new`出来的对象去调用的，而非静态的方法与变量则不需要使用`new`，它可以直接使用构造函数去调用就可以了

## class里面的`get/set`

在之前的`ES5`里面，我们可以通过`Object.defineProperty`去定义对象的访问器属性



```
//ES5的代码
let obj = {
    age: 15,
    firstName: "邓",
    lastName: "娜"
}
Object.defineProperty(obj, "isAdult", {
    enumerable: true,
    configurable: false,
    get: function () {
        return this.age >= 18 ? true : false;
    }
})
console.log(obj.isAdult);

Object.defineProperty(obj, "userName", {
    enumerable: true,
    configurable: false,
    get: function () {
        return this.firstName + this.lastName;
    },
    set: function (v) {
        if (typeof v == "string" && v.length > 0) {
            this.firstName = v[0];
            this.lastName = v.substr(1);
        }
    }
});
obj.userName = "蒋雨晴";
console.log(obj.userName);
```

现在在ES6里面，我们则使用另一种方式去定义



```
class Student {
    constructor() {
        this.age = 17;
        this.firstName = "邓";
        this.lastName = "娜"
    }
    //只有get没有set，所以只能取值
    get isAdult() {
        return this.age >= 18 ? true : false;
    }
    //取值的时候调用
    get userName() {
        return this.firstName + this.lastName;
    }
    //赋值的时候调用
    set userName(v) {
        if (typeof v == "string" && v.length > 0) {
```

```
            this.firstName = v[0];
            this.lastName = v.substr(1);
        }
    }
}

let stu1 = new Student();
// console.log(stu1.isAdult());  在没有加get之前，它是一个方法
console.log(stu1.isAdult);   //加了get以后，就是一个访问器属性了
stu1.userName = "蒋雨晴";
console.log(stu1.userName);
```

## extends关键字

在我们的ES5里面，我们让对象继承的方式有很多很多，但是都 或多或少有一些缺点



```
//定义了一个人的构造函数
function Person(userName) {
    this.userName = userName;
    this.sayHello = function () {
        console.log(`大家好，我叫${this.userName}`);
    }
}

//定义一个学生的 学生继承人类
function Student(userName, sex, age) {
    this.sex = sex;
    this.age = age;
    this.study = function () {
        console.log(`我今年${this.age}岁了，我可以上学了`);
    }
    this.__proto__ = Student.prototype = new Person(userName);
}
//定义一个女学生 继续学生
function GirlStudent(userName, age, sid) {
    this.sid = sid;
    this.makeup = function(){
        console.log(`我是女生，我要化妆，我的学生号${this.sid}`);
    }
    //开始继承
    this.__proto__ = GirlStudent.prototype = new Student(userName,"女",age);
}

// Student.prototype = new Person("邓娜");
let stu1 = new Student("邓娜", "女", 18);
// stu1.__proto__ = new Person("邓娜");
stu1.sayHello();
stu1.study();

let jyq = new GirlStudent("蒋雨晴",20,"001");
```

在上面的代码里面，表面看是没有任何问题的，但是它破坏了原型链

```
jyq instanceof GirlStudent ;//true
jyq instanceof Student;     //true
jyq instalceof Person;      //true
//但是
stu1 instanceof Student;    //false  问题在这里
jyq.constructor == GirlStudent; //false
```



**在ES6里面完美的解决了上面的问题**，ES6里面在使用继承的时候它使用`extends`做为关键字继承



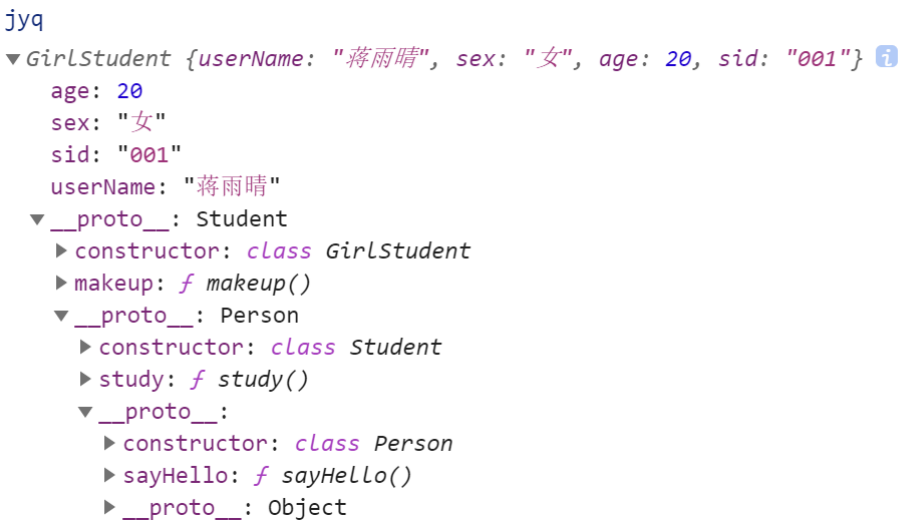
```
class Person {
    constructor(userName) {
        this.userName = userName;
    }
    sayHello() {
        console.log(`大家好，我叫${this.userName}`);
    }
}
// Student 现在通过extends继承了Person
class Student extends Person {
    constructor(userName, sex, age) {
        // super指向父级
        super(userName);
        this.sex = sex;
        this.age = age;
    }
    study() {
        console.log(`我今年${this.age}岁了，我可以上学了`);
    }
}
//GirlStudent继承Student
class GirlStudent extends Student {
    constructor(userName, age, sid) {
        super(userName, "女", age);
        this.sid = sid;
    }
    makeup() {
        console.log(`我是女生，我要化妆，我的学生号${this.sid}`);
    }
}

let stu1 = new Student("邓娜", "女", 18);
stu1.sayHello();
stu1.study();

console.log(stu1 instanceof Student);                    //true
console.log(stu1 instanceof Person);                     //true
console.log(stu1.__proto__ == Student.prototype);        //true

let jyq = new GirlStudent("蒋雨晴", 20, "001");
jyq.makeup();
jyq.study();
jyq.sayHello();
console.log(jyq instanceof GirlStudent);         //true
console.log(jyq instanceof Student);             //true
console.log(jyq instanceof Person);              //true
console.log(jyq.__proto__ == GirlStudent.prototype);//true
```

通过`extends`去继承的时候，我们发现所有的属性都在子级对象，而所有的方法都在父级对象上面




这么做的目的是为了实现`super`的关键字，`super`这个关键字会指向父级，指向父级的优点是为了实现另一个点**方法的重写**

## 方法的重写

方法的重写指的是子级对象要重写父级对象的方法，在我们之前开发的飞机大战的游戏里面，我们有一个方法`draw(ctx)`，这个方法后期在子对象里面进行了重写

**ES5的代码里面**

softeem · 杨标

```
function Person(userName) {
    this.userName = userName;
    this.sayHello = function () {
        console.log(`大家好，  我叫${this.userName}`);
    }
}

function Student(userName, sex) {
    this.__proto__ = new Person(userName);
    this.sex = sex;
    this.userName = "邓娜娜";
    //这一种情况叫方法的重写
    this.sayHello = function () {
        console.log(`大家好，我是一个${this.sex}学生`);
        this.__proto__.sayHello.call(this);
    }
}

let s1 = new Student("邓娜", "女");
s1.sayHello();
```

**ES6里面**

softeem · 杨标

```
class Person {
    constructor(userName) {
        this.userName = userName;
    }
    sayHello() {
        console.log(`大家好，  我叫${this.userName}`);
    }
```

```
}
class Student extends Person {
    constructor(userName, sex) {
        super(userName);
        this.sex = sex;
        this.userName = "邓娜娜";       //无论是调父级的属性，还是调子级的属性，都用this
                                        //属性不能用super
    }
    //重写父级对象的方法
    sayHello() {
        console.log(`大家好，我是一个${this.sex}学生`);
        //调的是父级的sayHello
        super.sayHello();
    }
}
let s1 = new Student("邓娜", "女");
s1.sayHello();
```

## 注意事项

### 第一步

softeem · 杨标

```
class Person {
    //没有写constructor,系统会自动帮你生成
    sleep() {
        console.log(`我在睡觉`);
    }
}
class Student extends Person {
    //没有写，系统会自动帮你生成
    study() {
        console.log(`我在学习`);
    }
}
let s1 = new Student();
```

在上面的代码当中，我们看到两个`class`都没有写`constructor`构造函数。如果系统发现你没有写构造函数`constructor`，它会自动的帮你生成一个无参数的构造函数`constructor`

**它生成的默认的构造函数到底是什么样的？**，它会自动的根据你有没有使用`extends`继承来决定是否调用`super()`

softeem · 杨标

```
constructor(){
    super();  //这里的super()就要注意了，如果你使用了extends继承，你就要写这个，没有继承则不要
    //super()是调用父级的构造函数
    //什么继承以后要调父级的构造函数呢？
    //根据正常的业务逻辑思维考虑，应该是先有父级对象，才有子级对象
}
```

### 第二步

softeem · 杨标

```
class Person {
    constructor() {
```

```
        //Person没有使用extends去继承，所以它不用使用super()
    }
    sleep() {
        console.log(`我在睡觉`);
    }
}

class Student extends Person {
    //如果我们主动了写了构造函数，则默认的那个构造函数就不存在了
    constructor(userName) {
        super();        //父级对象Person的构造函数
        this.userName = userName;       //this只能出现在super的后面
    }

    study() {
        console.log(`${this.userName}在学习`);
    }
}
let s1 = new Student("标哥哥");
s1.study();
s1.sleep();
```

---

最后再说一点，在继承上面，静态方法也是可以继承的，同时静态方法也支持重写，并且在重写的时候还可以通过`super`去调用父级

## 指数运算

在这前的ES5里面，如果我们想算某一个数的多少次方，是通过`Math`的方法来运算的，但是在ES6里面新增了一个方法

```
let a = Math.pow(2, 8);
console.log(a);

let b = 2 ** 8;
console.log(b);
```



指数运算符的结果一定是的个`number`类型，它的操作规则可以参考之前ES5里面的数字的操作规则

## 大整型数据类型

在之前的有ES5里面，如果我们使用数字`Number`类型进行运算的时候是容易出问题的，因为它的操作位数会不足，这个时候在ES6里面为了满足大数据的运算，所以我们多了一个数据类型`bigint`

```
let a = Math.pow(2, 53) ;
let b = Math.pow(2, 53) + 1 ;
console.log(a == b);//true
```



当我们去执行上面的判断过程的时候，我们发现a与b的值竟然是相等的，原因就是因为这个运算操作了JS的`Number`类型所能够运算的范围

为了弥补运算不足的问题，就推出了大整型

```
let temp = 1n;
console.log(typeof temp);      //bigint

//将之前的运算问题使用大整型去运算
let a = (2n ** 53n);
let b = (2n ** 53n + 1n);
console.log(a == b);  //false
console.log(a);
console.log(b);
```



## Reflect反射

反射是ES6里面新出的一种操作对象的方式，在之前的ES5里面，我们使用`Object`这个对象在操作对象，现在则又新出了`Reflect`

`Reflect`是一个静态的对象，它不能`new`，直接调用它自身的方法

1. `Reflect.get()`获取某一个对象的某一个属性
2. `Reflect.set()`设置某一个对象的某一个属性值
3. `Reflect.apply()`通过原来`apply`的方式调用一个方法
4. `Reflect.defineProperty`在某一个对象上面定义属性
5. `Reflect.deleteProperty`在某一个对象上删除某一个属性

上面是我们常用的一些操作方法，还有一些我截图如下

```
declare namespace Reflect {
    function apply(target: Function, thisArgument: any, argumentsList: ArrayLike<any>): any;
    function construct(target: Function, argumentsList: ArrayLike<any>, newTarget?: any): any;
    function defineProperty(target: object, propertyKey: PropertyKey, attributes: PropertyDescriptor): boolean;
    function deleteProperty(target: object, propertyKey: PropertyKey): boolean;
    function get(target: object, propertyKey: PropertyKey, receiver?: any): any;
    function getOwnPropertyDescriptor(target: object, propertyKey: PropertyKey): PropertyDescriptor | undefined;
    function getPrototypeOf(target: object): object;
    function has(target: object, propertyKey: PropertyKey): boolean;
    function isExtensible(target: object): boolean;
    function ownKeys(target: object): PropertyKey[];
    function preventExtensions(target: object): boolean;
    function set(target: object, propertyKey: PropertyKey, value: any, receiver?: any): boolean;
    function setPrototypeOf(target: object, proto: any): boolean;
}
```

总体而言，`Reflect`是我们换了一种方式，以第三人称的方式来操作我们原来的对象

## 代理Proxy

代理其实可以理解为现实中的经纪人模式，在程序员的眼中，代理其实就是全局的`get/set`

```
let huge = {
    userName: "胡歌",
    age: 38,
    sex: "男",
    money: 1000,
    address: "上海市浦东新区",
    Acting(actName) {
        console.log(`我现在正在拍${actName}这部戏`);
    }
}

let hugeProxy = new Proxy(huge, {
    get() {
        console.log("胡歌暂时不方便回答你的问题")
    },
```

```
    set() {
        console.log("胡歌不能接收你的东西");
    }
});


console.log(hugeProxy.userName);
hugeProxy.money = 5000;
```

**接下来我们使用ES6里面的代理实现属性私有化**



```
let hugeProxy = (() => {
    let huge = {
        userName: "胡歌",
        sex: "男",
        age: 38,
        _money: 5000,
        _address: "上海市浦东新区"
    }
    return new Proxy(huge, {
        get(target, p, receiver) {
            if (p.startsWith("_")) {
                return;
            }
            else {
                return target[p];
            }
        },
        set(target, p, value, receiver) {
            if (p.startsWith("_")) {
                return;
            }
            else {
                target[p] = value;
            }
        }
    })
})();

console.log(hugeProxy._address);
console.log(hugeProxy.age);
```

同时代理还可以实现属性的重定向



```
let gameConfig = (() => {
    let _gameConfig = {
        gameObjectContainer: {
            bg: 123,
            hero: 456,
            enemyList: [],      //敌机的数组
            bulletList: [],     //子弹的数组
            boomList: [],       //爆炸动画的数组
            buffList: []        //装道具的数组
        },
        maxEnemy: 10,          //屏幕上在出现的最大敌机数量
        score: 0,              //游戏得分
```

```
        isAddEnemy: true,      //是否允许添加敌机
        ctx: null,             //游戏的画笔
    }
    return new Proxy(_gameConfig, {
        get(target, p, receiver) {
            if (Object.keys(target.gameObjectContainer).includes(p)) {
                //处理
                return target.gameObjectContainer[p];
            }
            else {
                return target[p];
            }
        },
        set(target, p, value, receiver) {
            if (Object.keys(target.gameObjectContainer).includes(p)) {
                //处理
                target.gameObjectContainer[p].value;
            }
            else {
                target[p] = value;
            }
        }
    })
})();

console.log(gameConfig.bg);
console.log(gameConfig.hero);
console.log(gameConfig.maxEnemy);

gameConfig.enemyList.push("biaoege");
console.log(gameConfig.enemyList);
```

上面的问题就解决了我们之前在游戏当中所存在的一些问题，在之前的游戏里面，我们把所有的属性都放在了`gameConfig`下面，现在其实有部分属性我放在了`gameConfig.gameObjectContainer`下面

```
interface ProxyHandler<T extends object> {
    getPrototypeOf? (target: T): object | null;
    setPrototypeOf? (target: T, v: any): boolean;
    isExtensible? (target: T): boolean;
    preventExtensions? (target: T): boolean;
    getOwnPropertyDescriptor? (target: T, p: PropertyKey): PropertyDescriptor | undefined;
    has? (target: T, p: PropertyKey): boolean;
    get? (target: T, p: PropertyKey, receiver: any): any;
    set? (target: T, p: PropertyKey, value: any, receiver: any): boolean;
    deleteProperty? (target: T, p: PropertyKey): boolean;
    defineProperty? (target: T, p: PropertyKey, attributes: PropertyDescriptor): boolean;
    ownKeys? (target: T): PropertyKey[];
    apply? (target: T, thisArg: any, argArray?: any): any;
    construct? (target: T, argArray: any, newTarget?: any): object;
}
```

上面是代理所能够执行的所有操作，不仅仅是像我们上面那产只触发`get/set`，基本上对象的所有操作都可以被触发下来

## Symbol

这是ES6当的新数据类型[JavaScript当中第7种数据类型]，它是**全局唯一标识符**，不重复，与其它语言中的`GUID/UUID`非常像

> 全局唯一标识符（GUID，Globally Unique Identifier）是一种由算法生成的二进制长度为128位的数字标识符。GUID主要用于在拥有多个节点、多台计算机的网络或系统中。**在理想情况下，任何计算机和计算机集群都不会生成两个相同的GUID。**
>
> GUID 的总数达到了2^128（3.4×10^38）个，所以随机生成两个相同GUID的可能性非常小，但并不为0。所以，用于生成GUID的算法通常都加入了非随机的参数（如时间），以保证这种重复的情况不会发生。

通过上面的理解，我们可以得到一点

1. GUID这个东西是可以主动生成的
2. GUID不会重复

根据GUID的2个点，我们可以把它看成是我们ES6当中的一种symbol数据类型

在C#语言里面，如果想要生命全局唯一标识符，我们可以使用下面的代码

```
var a = Guid.NewGuid();          //a5baa104-7998-4ec9-9208-73be59fcf294
```



通过上面的代码，我们看到一个随机字符串，这个字符串一般情况下是不会重复的

在JavaScript当中，也有一个类似的方法，它实现的功能和`GUID`差不多，叫`symbol`；

```
var a = Symbol();       //这就生成了一个全局唯一标识符
//f35cddf4-5f74-11eb-954c-000ec6609035
var b = Symbol();       //这又生成了一个全局唯一标识符
//fdc88424-5f74-11eb-954c-000ec6609035
console.log(typeof a);       //"symbol"数据类型
console.log(a == b);    //false
```



在创建`symbol`，每次创建的都不相同的，这是它们的特点

我们在创建Symbol标识符的时候，还可以添加一个描述信息

```
var c = Symbol("标哥创建的");
var d = Symbol("标哥创建的");
console.log(c==d);       //false
```



虽然上面的`Symbol`具有相同的描述信息，但是它们仍然是不相同的

## Symbol的应用场景

```
let stu1 = {
    userName:"张三",
    sex:"男"
}
let stu2 = {
    hobby:"看书,睡觉",
    sex:"女"
}
//let obj = Object.assin({},stu1,stu2);
let obj = {
    ...stu1,
    ...stu2
}
```

> 在上面的代码当中，我们是合并了`stu1`与`stu2`两个对象为一个对象`obj`，这个时候我们发现它们有一个共同的属性`sex`， 这个时候会有什么现象产生呢?
>
> 这个时候因为属性名相同了，根据我们的键值对原则，后面的覆盖前面的，所以它的属性`sex`最终的值为**女**

有没有什么办法是这样的，让我们的对象在合并的时候属性名不被别人覆盖？？？

```
let stu1 = {
    userName:"张三",
    sex:"男",
    [Symbol()]:"haha"
}
let stu2 = {
    hobby:"看书，睡觉",
    sex:"女",
    [Symbol()]:"hello"
}
let obj = {
    ...stu1,
    ...stu2
}
console.log(obj)
```



```
> obj
< ▼{userName: "张三", sex: "女", hobby: "看书，睡觉", Symbol(): "haha", Symbol(): "hello"}
    hobby: "看书，睡觉"
    sex: "女"
    userName: "张三"
    Symbol(): "haha"
    Symbol(): "hello"
  ▶__proto__: Object
```

这个时候我们就看到了我们的两个`Symbol()`都是同时存在的，因为它们是不相同的，每个`Symbol()`都是一个唯一标识符

正是因为有了一个像这样的特点，所以，我们才可以使用`Symbol`做唯一的属性名（后期有大量的属性名是`Symbol`存在的）,也正是因为`Symbol`的特点，所以我们的一些特殊的属性名会以`Symbol`做为属性名，如`Symbol.iterator`等

## Symbol.for

如果在开发的过程当中，我们要使用相同事的`Symbol`值，则可以使用下面的方式来进行

```
let a = Symbol.for("aaaa"); //指定这个全局唯一标识符所生成的内容
let b = Symbol.for("aaaa");
let c = Symbol.for("aabb");
console.log(a == b); //true
console.log(b == c); //false
```



这一种方式就是根据一个具体的值来生成我们的Symbol数据类型（也就是根据具体的值来生成唯一的标识符）

## Object.getOwnPropertySymbols

如果想获取某一个对象的`Symbol`的属性名，普通的方法是不可以实现的

```
//Symbol做为属性名的时候，一定要使用中括号
var obj = {
    userName: "邓娜",
    age: 18,
    sex: "女",
    [Symbol()]: "hello"
}
Object.keys(obj); //["userName","sex","age"]
Object.getOwnPropertyNames(obj); //["userName","sex","age"]
Object.getOwnPropertySymbols(obj); //这里才可以得到Symbol
```



## 生成器函数

> 生成器函数叫`Generator`

生成器函数是一种特殊的函数类型，本质上面它也是一种函数，生成器函数调用执行以后，返回的是一个`Generator`的对象，这个对象有一个特点，它内部有一个`nex()`的方法

```
function abc() {
    //这就是一个函数
    //这个函数如果需要返回一个内容出去，我们可以使用return关键字
    return "hello";
}

let str = abc();
console.log(str);
```



在以前的函数里面，如果我们要返回一个内容出去，我们只能使用`return`去返回，但是有时候我们如果需要多次返回，怎么办呢?

那么，有没有一个思路是这样的，我一次给一个东西到外边，不停的去给，这就相当于我们之前所学习的迭代方法

例如数组里面的`forEach`每次都遍历一个值出去给外边的回调函数

```
let arr = ["a","b","c"];
arr.forEach(item=>{
    //这个item代表的就是里面的每一项，我每次拿到的都是其中的一个值
    //forEach会把里面的值一个一个的给我
    console.log(item);
});
```



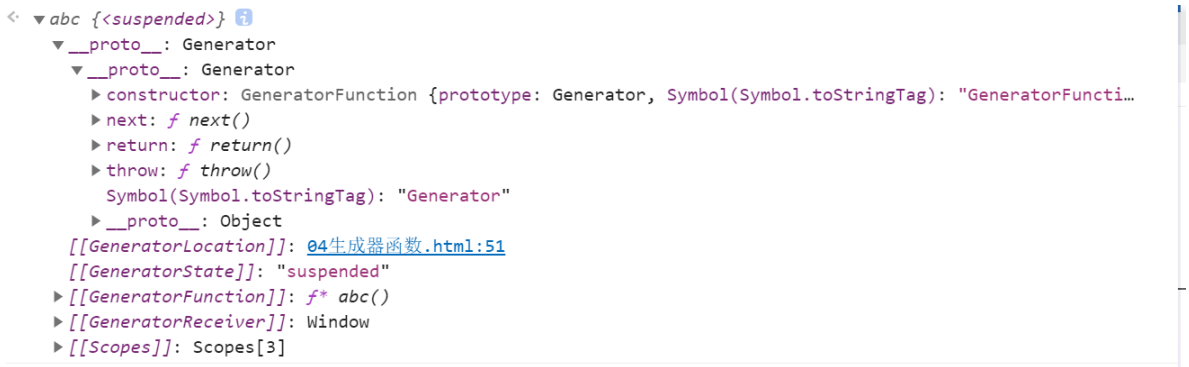
### 生成器函数的定义

```
 function* abc() {
     return "邓娜";
 }
let n = abc(); //这个n是一个生成器对象，
```



生成器函数执行以后返回的是一个生成器对象`Generator`，它的内部有一个方法叫`next()`。这个生成器对象也叫迭代器对象

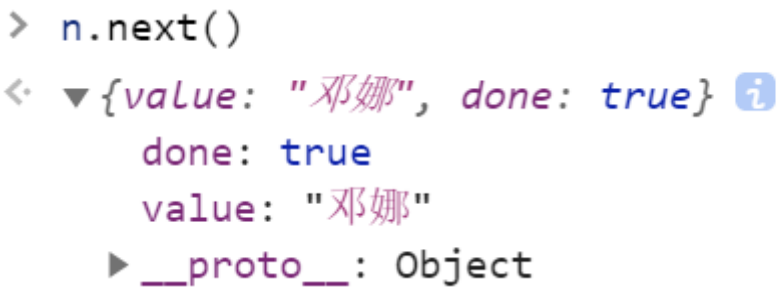
默认情况下，返回的这个生成器对象是暂停的，如果我们调用`next()`的方法，这个函数就会继续向后面执行，返回的也是一个对象




```
n.next();       //调用生成器对象的`next()`，让生成器对象由暂停状态进入下一步
```



上面的代码执行以后，返回的结果如下




在上面的结果上面，我们看到了两个属性

- `done:true`代表当前的代码已经执行完毕了【也就是执行到了最后】
- `value:"邓娜"`代表这次执行的结果返回的值是“邓娜”

**目前阶段来看，生成器函数有一个点它执行以后并不会立即执行函数体，处于一个“暂停supended”状态，直到我们调用next()的方法，才会继续执行，执行以后返回的有两个值，一个是done代表执行的结果，一个是value代表当前返回的值**

# 这么做到底有什么用呢？？？

我们刚刚都说过`return`在函数的内部只能返回一次，那么如果我现在希望返回多次呢？这个时候生成器函数会配合另一个关键字`yield`，它和`return`的作用很相似，都是用于返回一个值到外部的

**return和yield最大的区别就在于return一旦执行以后函数就退出，而yield在返回了内容以后就会让这个生成器处理“暂停”状态，而并不会结束**

```
function* abc() {
    //我想返回邓娜，蒋雨晴，易兰，徐大江
    //每次只返回一个值，怎么办呢？
    //如果使用return我们只能返回一个啊
    yield "邓娜";
    yield "蒋雨晴";
    yield "易兰";
    yield "徐大江";
}

let n = abc(); //调用生成器函数，返回生成器对象,这个对象默认是supended暂停状态
//而生成器对象的内部又有一个方法叫next()，让函数继续执行，直到碰取了yield就停下来，并返回yield的这个值所包含的对象
let a1 = n.next(); //{value: "邓娜", done: false}
```

```
console.log(a1);
// down:false说明没有完
let a2 = n.next();
console.log(a2);
let a3 = n.next();
console.log(a3);
let a4 = n.next();
console.log(a4);
let a5 = n.next();
console.log(a5);              //{value: undefined, done: true}
//最后面得到的a5里面的done:true代表我已经走完了，
```

我们现在通过这个`yield`与生成器函数让某一个函数内部多次返回了值过来，相当于一次一次的去拿值

上面的取值是一个一个的取值，与我们之前所学习的某些对象好像很像，如`Map`或`Set`，现在我们就用我们之前所学习的`Set`去试一下



```
function* abc() {
    yield "邓娜";
    yield "蒋雨晴";
    yield "易兰";
    yield "徐大江";
}
let n = abc(); //返回的是一个迭代器

let s1 = new Set(["邓娜", "蒋雨晴", "易兰", "徐大江"]);
let n2 = s1.keys(); //返回的也是一个迭代器!
let a1 = n2.next();
console.log(a1);        //{value: "邓娜", done: false}
let a2 = n2.next();
console.log(a2);
let a3 = n2.next();
console.log(a3);
let a4 = n2.next();
console.log(a4);
```

所以我们可以推断出来，`s1.keys()`它也是一个生成器函数，竟然是一个生成器的函数，那么，我们能否去借鉴`Set`的思维去理解生成器函数呢?



```
function* abc() {
    yield "邓娜";
    yield "蒋雨晴";
    yield "易兰";
    yield "徐大江";
}
let n = abc(); //返回的是一个迭代器

let s1 = new Set(["邓娜", "蒋雨晴", "易兰", "徐大江"]);
//如果我们一个一个去next就很麻烦，但是我们如果借鉴Set的思维看能够一次性就把值拿出来
/*
    for (let item of s1) {
        console.log(item);
    }
*/
//for...of的遍历是迭代器遍历，无序遍历，不需要索引.......
for (let item of n) {
```

```
    console.log(item);
}
```

我们再在再使用呢开运算符去测试一下

```
console.log(...s1); //展开了set集合
console.log(...n);  //这里也是可行的
```



所以这样我们的知识点就快慢慢的形成一个闭环了，我们之前所学习的展开运算符与`for...of`的无序遍历其实都是在内部调用`生成器函数`，如果你想让一个对象可以使用`for...of`或展开运算符，则内部必定会有生成器函数

## 迭代器

> 迭代的意思就是从一个对象内部一个一个的把内容拿出来

首先，我们要弄清楚一个问题，我们之前为什么说展开运算符`...`并不是所有的对象都可以使用，只有具备的了`Iterable`这个接口才可以？而学习了上面的生成器函数以后，我们又知道了一点，这个接口无非就是一个生成器函数，有了这个函数，我就可以迭代了

第一点，我们要先弄清楚什么是**接口**，以及**接口的概念是什么?**

接口：`interface`，接口的可以理解为规范，当一个对象实现某一个接口以后就必须要遵守这个接口的规范（我们也称之为具备这个接口的能力）

ES6里面有个迭代器的接口叫`Iterable`,实现了这个接就默认要实现这个接口的规范（你遵守了这个接口的规范，那么我就认为你具备这个接口能力）

`Iterbale`这个接口它主要定义的规范就是可以让元素使用`for...of`去遍历（能够使用for...of的就一定可以使用`...`的展开运算符）

所以判断一个对象能否使用`for...of`去或`...`展开运算符，只需要去判断这个对象有没有实现这个接口就行了，而在`JavaScript`里面，默认以下几个对象就实现了这个接口

1. `NodeList、HTMLCollection`
2. `Array`
3. `TypeArray`
4. `arguments`
5. `Set`
6. `Map`

所有迭代器默认都会有`next()`方法，这里不再细述了，这个方法可以让程序继续运行，直到遇到`yield`就暂停

我们可以让一个对象手动的去实现这个接口，这样这个接口不也就就可以使用`...`或`for...of`了吗

关键的问题就在于如果去实现这个接口（还是要之前我所说的一句话，具备某个接口，就要实现这个接口的规范，而`Iterable`这个接口的规范就是要在这个对象上面写一个`Symbol.iterator`这个方法）

## 实现`Iterbale`接口

```
let stu = {
    userName: "邓娜",
    age: 19,
    sex: "女",
    address: "湖北省武汉市",
    //只要有Symbol.iterator这个方法，就相当于实现了这个接口,注意它是一个生成器函数
    [Symbol.iterator]: function* () {
```

```
        //获取自己的属性值，一个一个的返回出去
        //获取自己的属性值，一个一个的返回出去
        let keys = Object.keys(this);   //首先获取到所有的属性名，但是获取不到Symbol
        let i = 0;
        while (keys[i] != undefined) {
            yield this[keys[i]];
            i++;
        }
    }
}
//我现在如果想使用for...of去遍历obj或在obj上面使用展开运算符...，应该怎么办呢
//console.log(...stu);              //必然会报错
//必须要让obj去实现这个接口
//如果要让它实现`Iterable`这个接口，则它必须有一个`Symbol.iterator`这个方法
for (let item of stu) {
    console.log(item);
}
console.log(...stu);
```

### 细节注意

1. `Iterable`是接口，以`able`结尾的代表的是能力的问题
2. `Iterator`是方法，它代表迭代器的方法

## 总结

我们先从概念上面来进行总结

1. `Symbol`是一个特殊的数据类型，而`Symbol.iterator`则是`Symbol`数据类型下面的一个特殊属性值
2. 迭代指的是从对象的内部一个一个拿出来值出来
3. 生成器它是函数的一种特殊叫法，它主要指的是定义函数的时候在函数名的前面有一个`*`这样的东西
4. 生成器函数执行以后，并不会立即执行代码体，它处于一个暂停的状态supended，它会返回一个迭代器，这个迭代有一个`next()`的方法，只有调用这个`next()`的方法函数才会真正的执行
5. 在生成器函数内部，如果碰到了`yield`，它会再次暂停运行，并且把`yield`后面的值返回出去；真到再次执行`next()`的方法
6. 迭代器其实是一个具备特殊名子的生成器函数，这个函数名叫`Symbol.iterator`
7. 所谓实现的`Iterable`接口，其实指的就是这个对象的内部有了`Symbol.iteratlr`这个生成的函数
8. 只要是实现了`Iterable`接口的就必然可以使用`for...of`以及`...`展开运算符

最后说一点：**迭代器，生成器函数，以及目前所学的`Symbol`数据类型都是为了解释for...of和展开运算符的**