ES6常用语法复习

一、ES6语法-var-const-let用法详解

1. 主要内容

- JavaScript 的作用域
- var 的局限性
- let 的用法详解
- const 的用法详解

2. Javascript的作用域

> **重点：Javascript只有函数作用域，没有块级作用域。**
>
> **也就是function里面定义的变量是有作用域的，if、for等代码块里面定义的变量是没有作用域的。**

```
if(true){
  //externalVal没有作用域
  var externalVal = "externalVal";
}
function domainTest(){
  var funVal ="funVal";
  //这里可以访问到externalVal和funVal
  console.log(externalVal + "----" + funVal);
}
domainTest()
if(true){
  console.log(externalVal); //可以访问到externalVal
  //console.log(funVal); 这一行访问不到变量，报错
}
```

没有代码块作用域

因为funVal定义在函数里面，函数是有作用域的

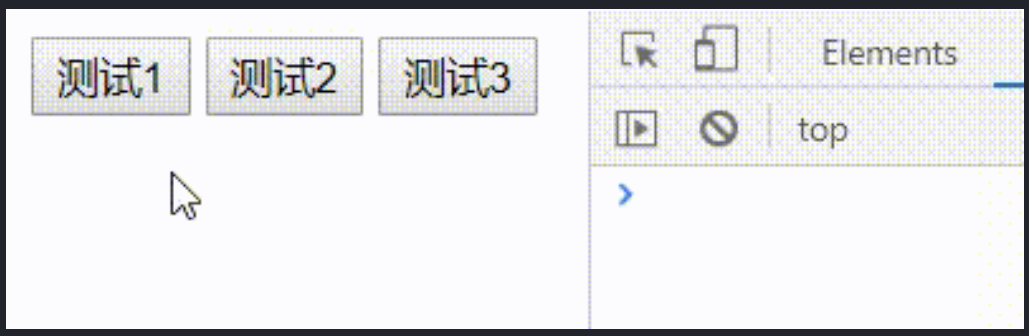
3. var的局限性

> **我们通过一个现象，来演示使用var定义变量的局限性。**

首先我们在html里面定义三个按钮，然后为三个按钮添加监听。

```
//使用var定义变量的局限性
var btns = document.getElementsByTagName('button')
for(var i = 0;i < btns.length;i++){
  btns[i].onclick = function(){
    console.log("点击了第" + i + "个按钮")
  }
}
```

测试发现，我们不期望的结果出现了，点击第一个按钮，打印“点击了第3个按钮”。



这是为什么呢?

大家想一想**作用域**，首先 **for代码块没有作用域**，当for循环绑定监听完成的时候，i是3。那么我们点击按钮的时候i也是3。

所以打印“点击了第3个按钮”。

那么我们该怎么解决这个问题呢?

我们需要使用**“闭包”**，也就是用一个函数把我们的操作包起来，让在它里面定义的变量或参数都有作用域。

```
for(var i = 0;i < btns.length;i++){
  (function(i){
    btns[i].onclick = function(){
      console.log("点击了第" + i + "个按钮")
    }
  })(i)
}
```

这样改完之后，我们再点击第一个按钮，打印“点击了第0个按钮”，没有问题。

4. let的用法详解

> **var的作用域问题是JS语言在最初设计时候的一个缺陷，使用闭包可以弥补这个缺陷。**

但有没有更优雅的解决方案呢？是有的，在ES6版本中，有了新的定义变量的方式let。

```
for(let i = 0;i < btns.length;i++){
  btns[i].onclick = function(){
    console.log("点击了第" + i + "个按钮")
  }
}
```

将var改为let，点击第1个按钮，打印“点击了第3个按钮”的情况就不会再出现。

因为let定义的变量自带块级作用域。

5. const的用法

> **当我们希望定义的变量不要被二次赋值的时候（也就是常量），使用const关键字来定义。**

```
const a = 50;
a = 100; //报错
const b; //报错，常量定义的时候必须赋值
```

二、ES6语法-对象的增强写法

1. 主要内容

- ES5的对象定义语法
- ES6定义对象的语法
- ES6字面量定义对象简写

2. ES5对象定义的语法

```
function Player(name,age){          定义对象
  this.name = name;
  this.age = age;
}
                          定义对象的成员函数
Player.prototype.toPrint = function(){
  console.log(this.name + "---" + this.age)
}
var player1 = new Player("james",35);
var player2 = new Player("kobe",39);
player1.toPrint()  //james---35
player2.toPrint()  //kobe---39
                                  调用成员函数
```

ES5中对象最令人觉得繁琐的就是构造函数、prototype(原型)、依靠原型链实现继承。

因为它使用的不是面向对象的语法，所以在使用过程中显得比较乱。

3. ES6对象定义的语法

在ES6语法中，JS引入了传统面向对象编程语言的语法(和Java非常类似)。

新的写法更符合面向对象编程的思想，也更加容易理解。

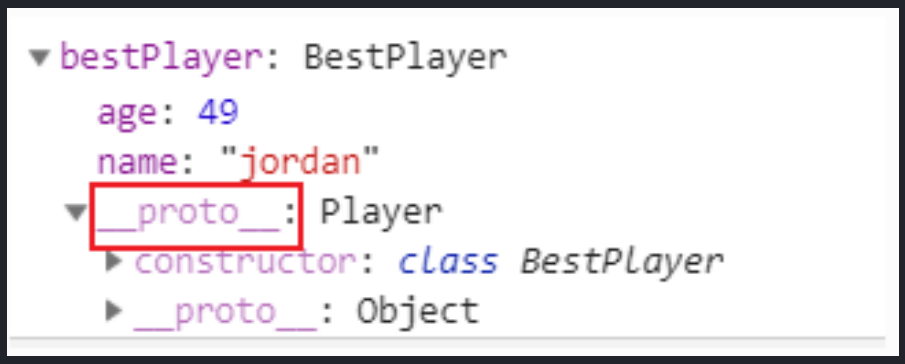
```
        定义对象
class Player{
  constructor(name,age){  定义构造函数
    this.name = name;
    this.age = age;
  }

  toPrint(){      定义成员方法
    console.log(this.name + "---" + this.age)
  }
}
var player1 = new Player("james",35);
player1.toPrint()  //james---35
```

- 引入calss关键字，用于定义对象
- 构造函数的方法名称是固定的，就叫constructor。这与Java等面向对象语言不同。
- 在对象定义(类)内，this关键字就代表当前实例对象。

```
class BestPlayer extends Player{    extends关键字实现继承
  constructor(){
    super()        super关键字调用父类构造方法
    this.name = "jordan"
    this.age = 49
  }
}
let bestPlayer = new BestPlayer();
bestPlayer.toPrint()  //jordan---49
```

- 通过class关键字实现了类的继承
- 通过super()关键字调用父类构造函数，如果不传参数，默认是undefined
- super()方法必须显示调用，否则子类找不到this指针。

实际上 JS 对象的继承原理，仍然是依靠原型链来实现的。只是ES6给了我们语法糖，使得更好理解更易使用。



4.ES6字面量定义对象简写

创建对象除了使用构造方法，还可以使用字面量的方式来创建一个对象。这种写法会经常用到。
在ES5中的写法:

```
let name = "curry"
let age = 33
let player3 = {
  name :name,
  age : age,
  toPrint:function(){
    console.log(player3) //{name: "curry", age: 33}
  }
}
player3.toPrint() //{name: "curry", age: 33, toPrint: f}
```

在ES6语法中，可以简写为

```
let name = "curry"
let age = 33
let player3 = {
  name,          注意这里等同于name: name
  age,
  toPrint(){     这里可以省略function关键字
    console.log(player3) //{name: "curry", age: 33}
  }
}
```

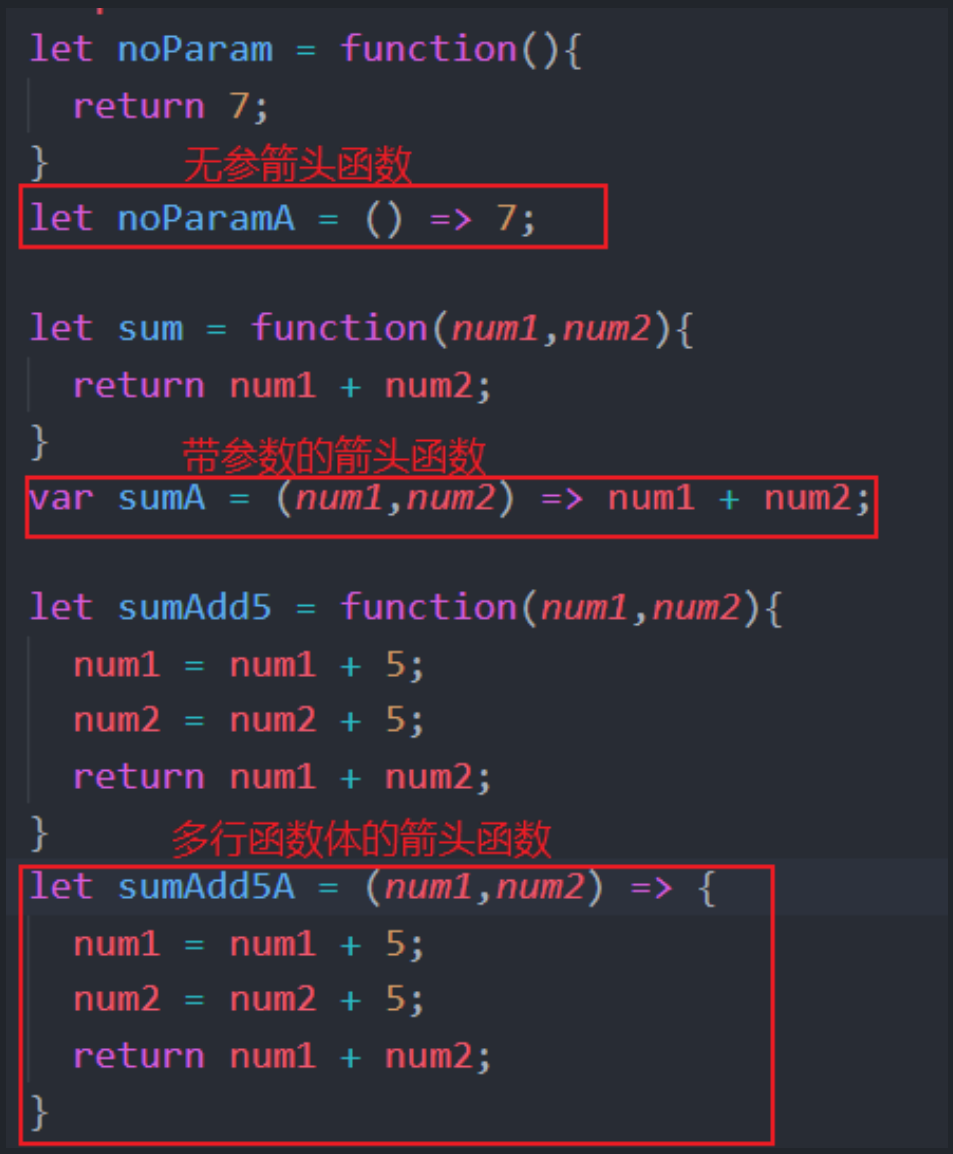
三、ES6语法-箭头函数与this指针

1. 主要内容

- 箭头函数简写形式
- 对象中的this指针
- 全局环境及函数中的this指针
- this指针使用过程中存在的一个问题，如何用箭头函数解决

2. 箭头函数简写形式

下面代码是普通函数定义与箭头函数的对比：



可以看到，箭头函数的写法更为简便，箭头左侧定义参数，箭头右侧定义函数体。当函数体为一行内容时，return关键字可以省略。

3. 对象中的this指针

```
        定义对象
class Player{
    constructor(name,age){  定义构造函数
        this.name = name;
        this.age = age;
    }

    toPrint(){      定义成员方法
        console.log(this.name + "---" + this.age)
    }
}
var player1 = new Player("james",35);
player1.toPrint()  //james---35
```

在对象中，this指针指向的就是对象本身。this可以引用对象的属性和方法。

4. 全局环境及函数中的this指针

1） 全局环境中的this指针

```
console.log(this === window); // true
console.log(this.document === document); // true
```

全局环境中，this指针指向window全局对象。

2） 函数中的this指针

谁调用函数，函数内的this指针就指向谁

我们平时在使用 JS 函数时，在全局环境下实际是 window 对象调用函数。

```
function test1(){
  return this;
}          上面的写法实际是下面写法的简写，我们平时经常忽略掉window全局对象的存在
console.log(test1() == window);  //true
console.log(window.test1() == window);  //true

function test2(){
  "use strict"  //使用严格模式
  return this;
}
console.log(test2() == undefined);  //true
console.log(window.test2() == window);  //true
```

在函数的内部this指针的指向有两种可能:

- 在严格模式下，this指针为undefined。严格模式，是js更严谨、更安全的一种发展方向的体现。
- 在非严格模式下，this指向window全局对象。

5. this指针使用过程中存在的一个问题

下面的代码的执行结果是什么？（**注意我们加上了一个setTimeout定时函数**）

```
class Player{
  constructor(nickname,age){
    this.nickname = nickname;
    this.age = age;
  }

  toPrint(){
    setTimeout(function(){
      console.log(this.nickname + "---" + this.age)
    }, 1000);
  }
}
let player1 = new Player("james",35);
player1.toPrint()
```

我们期望的执行结果是“james---35"，但是实际的执行结果是“undefined---undefined”。
上文我们说过，**谁调用函数，函数内的this指针就指向谁**。那么哪个对象调用了setTimeout的定时函数呢?
是player1么？**不是，是window**。window没有nickname和age属性，所以打印“undefined---undefined”。

```
toPrint(){
  window.setTimeout(function(){
    console.log(this.nickname + "---" + this.age)
  }, 1000);
}
```

那么我们怎么解决这个问题呢？， 下面三种方法都可以打印我们期望的结果：“james---35"

- 额外定义this指针的替身

```
toPrint(){
  let _this = this
  window.setTimeout(function(){
    console.log(_this.nickname + "---" + _this.age)
  }, 1000);
}
```

- 使用`bind(this)`

```
toPrint(){
  window.setTimeout(function(){
    console.log(this.nickname + "---" + this.age)
  }.bind(this), 1000);
}
```

- 最简单的方式：箭头函数

```
toPrint(){
  window.setTimeout(() =>{
    console.log(this.nickname + "---" + this.age)
  }, 1000);
}
```

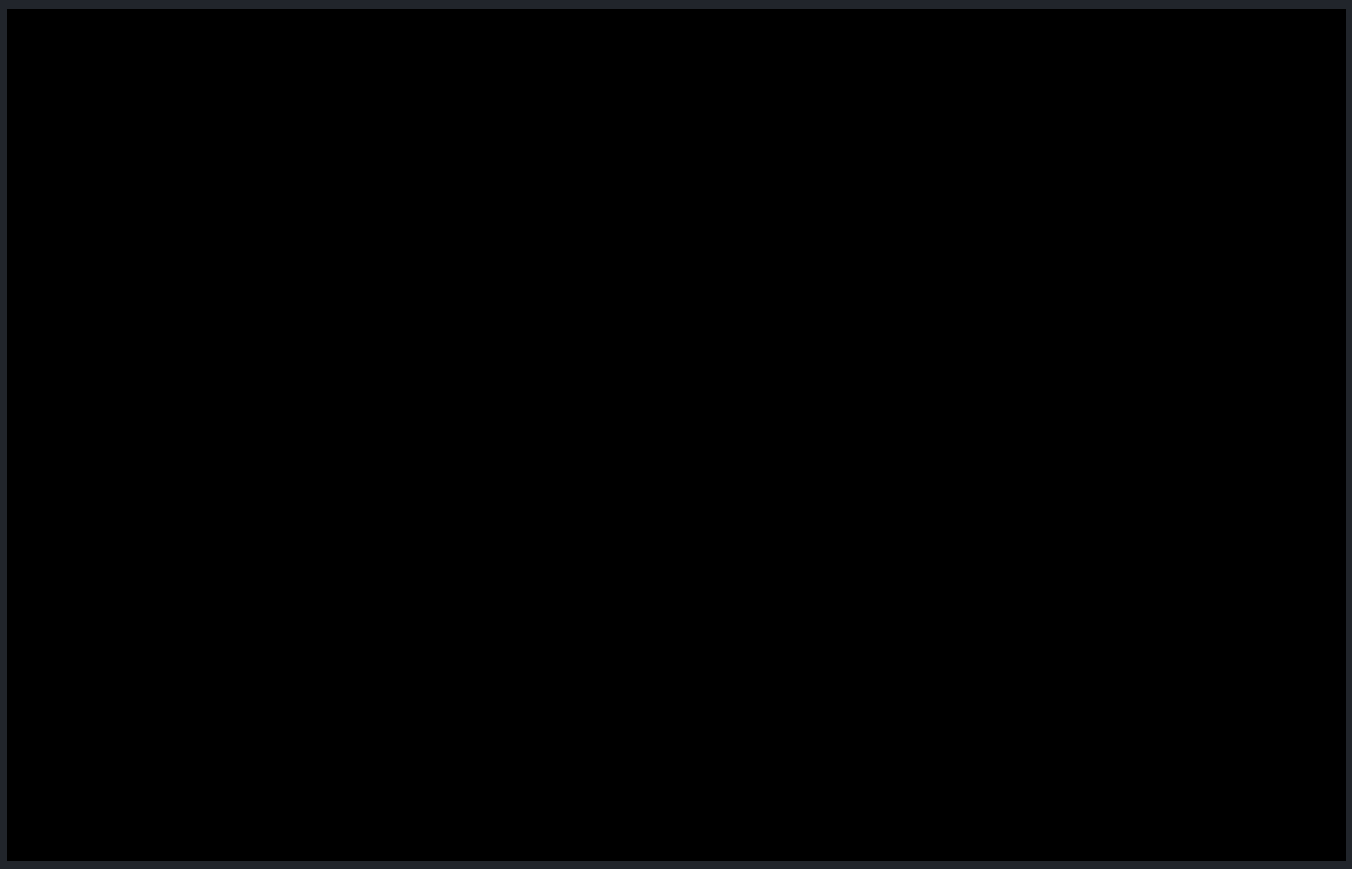
四、ES6语法-变量的解构赋值

1. 主要内容

- 数组的解构赋值
- 对象的解构赋值

- 字符串的解构赋值
- 解构赋值在实际应用中常用例子

2. 数组的解构赋值



数组的解构赋值的基本要求就是：**等号左侧赋值变量数组和等号右侧被解构的数组**，能够在模式上正确匹配。

3. 对象的解构赋值

```
let {age:age,name:name} = {name:"james",age:35}
//对象的解构赋值,变量名对上即可，顺序不重要
//简写形式:  let {age,name} = {name:"james",age:35}
console.log(name + "----" + age) //james-35
```

不但可以解构对象里面的属性，还可以解构对象的成员方法。

比如大家都知道Math对象的三个方法：sqrt、sin、cos分别用于计算平方根，正弦，余弦。

```
//对象方法的解构
let {sqrt ,sin ,cos} = Math
console.log(sqrt(4))  //2
```

4. 字符串的解构赋值

```
let [a1,a2,a3] = "curry"
console.log(a1) //c
console.log(a2) //u
console.log(a3) //r
```

5. 解构赋值在实际应用中常用例子

五、JS数组操作

1. 数组的创建方式

- 通过构造函数创建数组

```
var players = new Array();
let  players = new Array(20);    //length为20的数组
let players = new Array("curry","james","kobe");  //创建带有三个初始化项的数组
//关键字new可以被省略
let players = Array();
let players = Array(20);
let players = Array(20).fill(0);  //创建并填充一个20项初始值都为0的数组
let players = Array("curry","james","kobe");
```

- 通过字面量创建数组(推荐)

```
let players = [];
let players = ["curry","james","kobe"];     //创建带有三个初始化项的数组
```

- 通过Array.of()函数

```
let players = Array.of("curry", "james", "kobe")
console.log(players);  //["curry", "james", "kobe"]
```

2. 数组长度变化的影响

2.1 数组的length属性不是只读的，可以修改

这与绝大多数编程语言都不一样，那么修改了length属性有什么影响呢?

```
let players = ["curry","james","kobe"];
//原始长度是3，将数组长度修改为2，相当于删除了末尾的一项。因此可以使用该属性完成数组末尾数据删除功能
players.length = 2;
console.log(players[2])    //数组下标是2（第三项，0开始），打印结果是undefined
```

如果，将数组的length设置大于当前数组的项目，新加的每一项也都是undefined。

2.2 超过数组长度赋值的影响

```
let players = ["curry","james","kobe"];     //创建带有3个初始化项的数组
players[7] = "jordan";    //设置数组中第8项
console.log(players.length);     //输出结果为8
console.log(players[5])    //undefined
```

- 超出数组长度赋值，数组会自动扩充到指定项的长度。没有被赋值的项，为undefined

3. 如何检测一个数组的类型

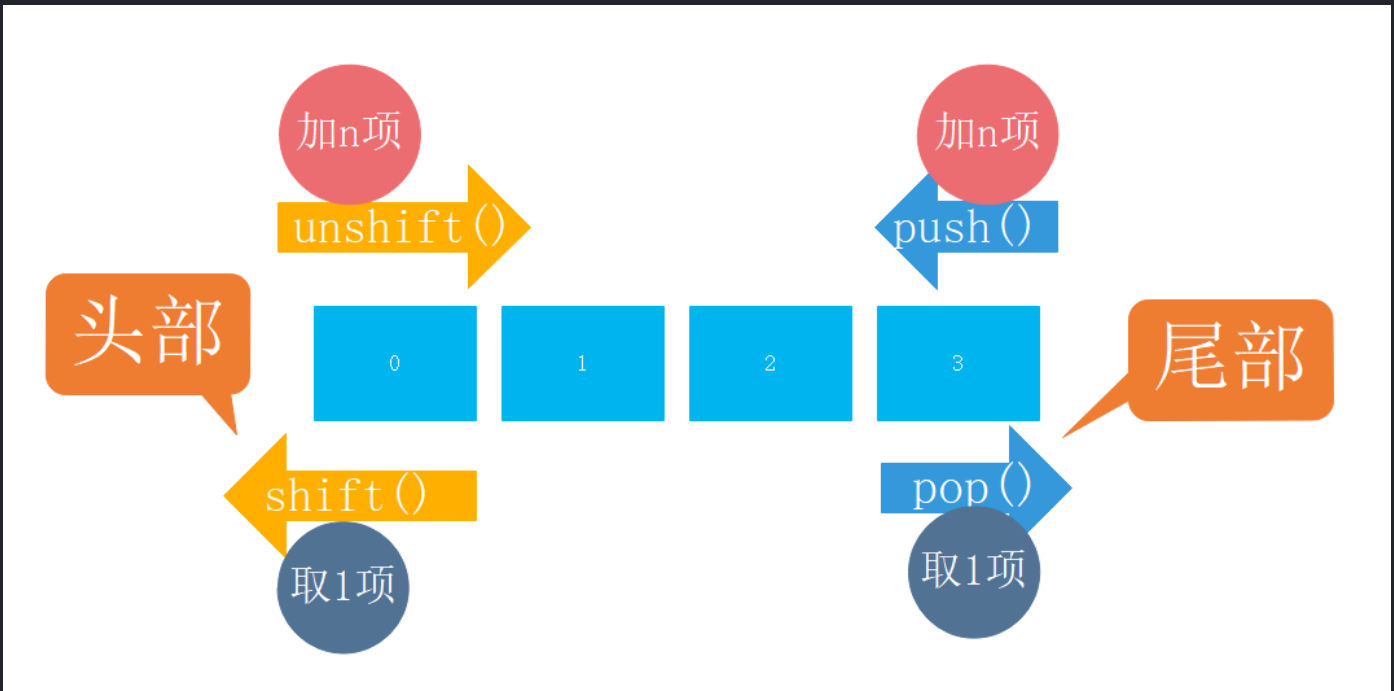
```
console.log(Array.isArray(players))       //true，ES5方法，推荐使用
console.log(players instanceof Array)     //true，多个js前端框架共用环境下可能有问题
```

4. 数组转字符串

```
let  players  = ["curry","james","kobe"]; //创建带有三个初始化项的数组
console.log(players.toString());          //curry,james,kobe。默认使用逗号分隔
console.log(players.toLocaleString());    //curry,james,kobe
console.log(players.join("&"));    //curry&james&kobe,如果想自定义分隔符号使用join
```

- toString()和toLocaleString()通常输出是一致的。但是当我们数组元素是js对象的时候，对象可以自定义这两个函数。locale通常指本地化。例如我们可以自定义这两个函数：toString返回值"kobe"，toLocaleString返回值"科比"。

5. (重点)如何用数组实现“栈”和“队列”



要实现栈或者队列我们得先熟悉下面的四个方法：

方法	作用位置	功能	例子
push	数组尾部	向数组尾部加n项	players.push("adu","wade")
pop	数组尾部	从数组尾部取1项，并在数组中删除该项	players.pop()
unshift	数组头部	向数组头部加n项	players.unshift("adu","wade")
shift	数组头部	从数组头部取1项，并在数组中删除该项	players.shift()

> **如何实现“队列”和“栈”数据结构?**

队列：先进先出，所以可以用push()和shift()方法配合实现队列，也可以使用unshift()配合pop()
栈：先进后出，后进先出。所以可以使用push()和pop()来实现。也可以使用unshift()和shift()来实现，但从语义的角度用前者更好。

6. 数组排序

6.1.基本数据类型排序

```
let  values  =  [1,2,5,10,19]
values.reverse()  //数组倒序方法
console.log(values.toString()) //数组倒序：19,10,5,2,1
values.sort()   //数组排序方法
console.log(values.toString()) //数组正向排序：1,10,19,2,5
```

大家看到上面代码中最终的排序结果是[1,10,19,2,5]。而我们期望的结果是[1,2,5,10,19],这是因为sort()方法默认将每一项 toString()之后在比较，字符串1,10,19的第一位都是1，所以小于2和5.
那么我们怎么让它正确排序呢？我们可以自定义比较规则，即：自定义比较函数

```
function  compare(value1,value2){   //定义比较规则的函数
    return  value1  -  value2  //返回值三种可能，0，大于0，小于0
}
values.sort(compare)     //使用比较规则
console.log(values.toString()) //排序结果：1,2,5,10,19
```

排序规则函数，通过返回0或者大于0或者小于0的数，来影响排序结果。

6.2. 比较对象

```
let players = [{name:"james",age:36},
               {name:"curry",age:31},
               {name:"kobe",age:39}]
function comparePlayer(player1,player2){
  return player1.age  - player2.age
}
players.sort(comparePlayer)
console.log(players)
```

结果（按年龄排序）：

```
▶ 0: {name: "curry", age: 31}
▶ 1: {name: "james", age: 36}
▶ 2: {name: "kobe", age: 39}
```

7. 数组的合并与剪切

- concat可以合并一项，也可以合并一个数组。所以可以用它向数组末尾添加项。

```
let players = ["curry","james","kobe"];
let players2 = players.concat("jordan",["拉里伯德","魔术师"])
console.log(players2)  //["curry", "james", "kobe", "jordan", "拉里伯德", "魔术师"]
```

- slice可以剪切一个数组的元素成为一个新的数组

```
//数组下标从0开始
//从第2项开始剪切到最后，["james", "kobe", "jordan", "拉里伯德", "魔术师"]
console.log(players2.slice(1))
//从第2项开始剪切到第5项，不包含第5项。  ["james", "kobe", "jordan"]
console.log(players2.slice(1,4))
```

8. 数组元素删除、插入、替换

```
let players = ["curry","james","kobe"];
//从下标为0的项开始，删除一项。
players.splice(0,1)
console.log(players)  // ["james", "kobe"]
//从下标为1的项开始，删除0项，插入2项。相当于插入操作
players.splice(1,0,"jordan","魔术师")
console.log(players) //["james", "jordan", "魔术师", "kobe"]
//从下标为1的项开始，删除1项，插入1项。相当于替换操作
players.splice(1,1,"杜兰特")
console.log(players) //["james", "杜兰特", "魔术师", "kobe"]
```

这个方法相对复杂，如何记忆这个方法？不要分别去记忆删除、插入、替换的实现。
记3个参数，从下标为(参数一)开始，删除(参数二)个元素，插入(参数三)个项。参数三可以是多个，并且是可选。
通过对三个参数的组合来实现删除、插入、替换操作。

9. 数组成员的查找

```
let players = ["curry","james","kobe","james","curry"];
```

```
//这两个方法在找不到数据时，返回-1
console.log(players.indexOf("james"))       //1,从数组头部查找数据返回下标位置
console.log(players.lastIndexOf("james"))   //3,从数组末尾查找数据返回下标位置
//查找符合条件的第一个数组成员,找不到返回undefined
let findOne = players.find(function(value,index,arr){
return value.includes("cu")
})
console.log(findOne)    //curry,curry包含cu
//查找符合条件的第一个数组成员下标，找不到返回-1
let findOneIndex = players.findIndex(function(value,index,arr){
return value.includes("cu")
})
console.log(findOneIndex)    //0,curry包含cu，第一个元素下标是0。
```

- 单纯的通过数据遍历比对，查找数组成员下标，用indexOf和lastIndexOf方法会更合适
- 如果需要更复杂复杂的规则查找数组成员，用find和findIndex方法更合适
- 上文中includes方式是ES6新方法，用于判断字符串的包含关系，返回值是布尔类型
- find和findIndex方法参数是：**当前的值，当前的数组成员下标，原数组**

上文的find可以简写为:

```
let findOne = players.find((value,index,arr) => value.includes("cu"))
let findOne = players.find((value) => value.includes("cu")) //只传第一个参数
```

ES6语法，箭头函数。**箭头左侧是方法参数，箭头右侧是方法体，return关键字可以省略**。

10. (重点)for-in与for-of遍历数组

遍历数组

```
let players = ["curry","james","kobe"];
for(index in players){
  console.log(index + ":" + players[index])
}
for(player in players){
  console.log(player)
}
```

```
0:curry
1:james
2:kobe
0
1
2
```

当实际业务使用不到数组下标时，使用for-of遍历更方便

11. 遍历数组keys()、values()和entries()

```
let players = ["curry","james","kobe"];
for(let index of players.keys()){
console.log(index) //打印是0,1,2
}
for(let elem of players.values()){
console.log(elem)   //打印是"curry","james","kobe"
}
for(let [index,elem] of players.entries()){
console.log(index,elem)   //打印如下
}
//0 "curry"
//1 "james"
//2 "kobe"
```

keys()、values()和entries()分别用于遍历数组的键(下标)、值、键值对。

12. (重点)数组迭代判定

```
let players = [{name:"james",age:36},
               {name:"curry",age:31},
               {name:"kobe",age:39}]
//every方法，判断是否数组每一个对象的年龄都大于30
let isGt30 = players.every(function(value,index,arr){
  return value.age > 30
})
console.log(isGt30)  //true
//some方法，判断是否数组中有至少一个对象的名字包含cu
let isContainCu = players.some(function(value,index,arr){
  return value.name.includes("cu")
})
console.log(isContainCu)  //true
```

迭代方法的三个参数：**当前的值，当前的数组成员下标，原数组**
只要理解了英文单词的语义，这两个方法不难记忆

- every，每一个
- some，存在一些

ES6箭头函数的简写形式如下：

```
let isGt30 = players.every((value) => value.age > 30)
let isContainCu = players.some((value) => value.name.includes("cu"))
```

> **想一想，如果你不知道every和some方法，你该怎么做？麻不麻烦？**

13. (重点)数组成员的迭代处理

```
let players = [{name:"james",age:36,champions:3},
               {name:"curry",age:31,champions:3},
               {name:"kobe",age:39,champions:5}]

players.forEach(function(value,index,array){
  //这个方法没有返回值，可以在这里执行一些操作，比如渲染一个table
  console.log(`<tr>${value.name}</tr>`)  //这里使用了ES6语法，避免字符串拼接
})
```

结果：

```
<tr>james</tr>
<tr>curry</tr>
<tr>kobe</tr>
```

14. (重点)数组的过滤与归并计算

```
//过滤数组中所有年龄大于30的球员，返回值也是一个数组
let ageGt30Players = players.filter(function(value,index,array){
  return value.age > 30
})
console.log(ageGt30Players) //数组所有球员年龄都大于30，所以ageGt30Players = players
//将数组的中所有人的年龄都减小5岁
let young5Plauers = ageGt30Players.map(function(value,index,array){
  return {name:value.name, age:value.age - 5, champions:value.champions}
})
console.log(young5Plauers)  //球员对象的年龄变为31,26,34
//计算数组中球员，总冠军的数量
let championsNums = young5Plauers.reduce(function(prev,value,index,array){
  return prev + value.champions
},0)
console.log(championsNums)  //11,三位球员的总冠军数量
```

- filter方法用于过滤数组的成员，满足条件的成员组成一个新的数组返回
- map方法用于对数组成员处理，返回一个处理完成之后的数组
- reduce用于对数组元素进行归并计算，和其他数组迭代方法不同，它的function有四个参数。其中第一个参数比较特殊，表示上一次迭代计算的结果。
- reduce除了迭代函数function，还有一个参数为初始值，第一次计算时候prev=初始值。我们这里设置为0

上面那一堆，使用ES6语法，可以简写为：

```
let championsNums = players.filter(value => value.age > 30)
                           .map(value => {return {name:value.name, age:value.age - 5, champions:value.champions}})
                           .reduce((prev,value) => prev + value.champions ,0)
```

15. Array.from()的用法

Array.from()可以将类似数组的对象和可比遍历的对象(如:ES6的Set)转换为数组

```
let playersObje = {'0':"curry",'1':"james",'2':"kobe",length:3};
let players = Array.from(playersObje);
console.log(players)   //["curry", "james", "kobe"]

let playersSet = new Set(['curry','james','curry'])
players = Array.from(playersSet);
console.log(players) //["curry", "james"],Set可以去重
```

16. copyWithIn()的用法

copyWithIn()用于将数组内的指定位置的成员，复制到其他位置（覆盖原有项）。

```
//输出结果为[3, 4, 2, 3, 4]
console.log([0,1,2,3,4].copyWithin(0,3,5))
```

- 第一个参数是target，复制到的位置
- 第二个参数是从哪个位置开始复制
- 第三个参数是到哪个位置结束复制

所以上面例子是将3,4复制到数组下标为0的位置，所以最终于是[3, 4, 2, 3, 4]

17. 数组推导与生成器推导

目前绝大部分浏览器都已经不再兼容。

六、Promise语法详解

1. 异步操作与同步操作

用白话举个例子：

- 同步：好比你给别人打电话，电话里面沟通立刻就有响应，这就是同步的。
- 异步：你给别人发短信，别人不一定马上回复。而是一段时间之后才有回复。

此时，对于发起异步操作的人，有三种选择：

- 第一种是一直看着短信，等待回复。你什么也做不了，就是等着。这就是阻塞IO
- 第二种是每隔一段时间去查看一下对方是否回复。这是非阻塞IO
- 第三种就是你给对方留言中加上了回调：“当你看到短信时给我打一电话”，这样你就去干别的事了。第三种是我们在js异步操作中常用的方法。

2. 传统异步操作中的问题

> **Promise是用于处理异步操作及异步操作回调的结构优化语法，在ES6中被提出。**

在ES6之前，当我们在异步回调函数结果中获取数据，再次发起异步操作。周而复始，可能会出现一种被称为**回调地狱**的代码结构：

- 这种结构首先缩进很难排版及阅读，一不小心就将代码写错了地方
- 另外这种结构，无法将异步操作与异步操作回调结果解耦。在一些结构化比较好的js程序中，可能发起操作的是A模块，做异步结果处理的是B模块。

Promise就是为了解决上述问题，应运而生的。

3. Promise的三种状态

大家一定要有一个概念，**JS 函数的参数不仅可以是字符串、数字等基本类型，也可以是函数。**

- Pending (等待中)：Promise 的初始状态，异步操作进行中。如：网络请求正在处理，读写文件正在进行。
- Fulfilled (已实现)：异步操作已经实行成功。
- Rejected (已拒绝)：异步操作过程执行失败。

4. Promise基础语法

- Promise异步操作主要由两部分组成，一是New Promise发起异步操作，而是then函数处理异步操作结果
- Promise异步函数有两个参数，这两个参数也是函数，resolve和reject。
- 当resolve函数被调用，表示异步操作成功。then函数的第一个函数参数方法被回调。可以传递一个对象作为成功处理的结果参数，如图中的data={}
- 当reject函数被调用，表示异步操作失败。then函数的第二个函数参数方法被回调。同样可以传递一个对象作为失败处理的结果参数，如图中的err= Error对象。

这样做的好处就是：异步操作与异步操作结果回调就被解耦了。我们完全可以把new Promise放到A模块，并将p变量导出；B模块导入变量p，使用p.then进行回调结果处理。

5. 链式调用

当我们的异步操作中还有异步操作，为了避免回调地狱的代码格式，我们应该使用链式调用。或者异步操作中有同步操作，也可以使用链式调用。所谓链式调用必须保证：函数的调用者与函数的返回值都是同一个类型，才能链式调用。比如then函数的调用者和返回值都是Promise类型 。

- then函数如果只有一个参数，那就是异步操作成功的回调函数。then函数的第二个函数参数是可选的。
- 链式调用的catch函数的参数，是异步操作失败的回调函数。

```
var p = new Promise(function(resolve, reject) {
    // 异步操作
    if(异步操作成功) {
        resolve('Success');
    }
    else {
        reject('Failure');
    }
});

p.then(function(data) {   //处理异步操作成功数据
    //处理异步操作的结果
    return new Promise((resolve) =>{
        resolve( data + "xxx1")
```

```
    })
}).then(function(data) {   //处理异步操作成功数据
    //处理异步操作的结果
    //上面的那一段then函数的简化写法
    return Promise.resolve( data + "xxx2")
}).then(data => {    //处理异步成功操作数据。简化为箭头函数
    //处理异步操作的结果
    //上面的那一段then函数更简化的写法
    return data + "xxx3"
}).catch(function(error) {
    /* error */
})
```

上面关于Promise对象的构建，并调用resolve方法，写了三种形式。这三种形式效果是一样的。后两种方式进行了不同程度的简写。简单起见，成功回调函数的参数data就写成了字符串，字符串也是一种对象。
同样，下面三种构造Promise对象，并调用reject方法的三种形式，也是一样的效果。

```
//方式一:
return new Promise((resolve.reject) =>{
   reject({errorMsg:""})
})
//方式二:
return Promise.reject({errorMsg:""})
//方式三:
throw {errorMsg:""}
```

6. Promise的all方法

我们可能会遇到这样的需求，A、B、C三个异步操作，必须这三个操作都完成了才能进行下一步操作。在没有promise的时候，我们通常定义三个变量aflag、bflag、cflag初始值为false，当A执行完成时置aflag=true。以此类推。判断三个flag都为true，才执行下一步的操作。

有了Promise的all方法，这一切就变得顺理成章，我们再也不用判断flag了。代码如下：

```
var p1 = new Promise((resolve) => {
        resolve({msg:"1"})
}),p2 = new Promise((resolve) => {
        resolve({msg:"2"})
}),p3 = new Promise((resolve) => {
        resolve({msg:"3"})
});
Promise.all([p1, p2, p3]).then(function (results) {
    console.log(results);  // [{msg:"1"},{msg:"2"},{msg:"3"}]
});
```

当p1、p2、p3三个异步操作都完成的时候，才执行then中的函数。then函数的参数results是一个数组，results[0]是p1异步执行成功的结果数据。以此类推。

7. 什么时候用?

在开发过程中，**当你想要获取异步操作的处理结果的时候，使用Promise!**

> **下面是一段NodeJS连接mysql数据库的代码，不要慌，没让你去学NodeJS。**
>
> **NodeJS也是 Javascript，当成Javascript代码看就可以。**

```
getTableInfo() {
    return new Promise((resolve, reject) => {
        //连接数据库的结果是一个回调函数，回调函数是异步的，我想获取连接结果
        this.connection.connect(err => {     //lambda写法的回调函数
            if(err) reject("数据库连接失败，请检查数据库连接！")
        });

        let sql = ' 定义一个SQL';

        //数据库查询结果是一个回调函数，回调函数是异步的，我想获取查询结果数据
        this.connection.query(sql, this.dbName,
            function (error, results, fields) {    //正常写法的回调函数
                if (error) {
                    reject(error);    //查询数据出错
                } else {
                    resolve(JSON.parse(JSON.stringify(results)))   //查询成
功，返回查询结果
                }
            }
```

```
        )
        //数据库连接关闭结果是一个回调函数，回调函数是异步的，如果连接关闭失败我需要得到响应
        this.connection.end(function (err) {
            if(err) reject("数据库关闭失败！")
        })
    })

}
```

结果处理

```
mysqlInfo.getTableInfo().then(results => {
   //将results查询结果渲染到页面上
}).catch(error =>{
    //给用户一个错误提示
})
```