ES6那么多那么多特性，我们真的需要全部都掌握吗？秉着二八原则，掌握好常用的、有用的这个可以让我们的开发快速起飞。接下来我们就聊聊ES6那些可爱的新特性吧。

1.变量声明const和let

在`ES6`之前，我们都是用`var`关键字声明变量。无论声明在何处，都会被视为声明在**函数的最顶部**(不在函数内即在**全局作用域的最顶部**)。这就是函数**变量提升**例如:

```
function aa(flag) {
   if(flag) {
      var test = 'hello man'
   } else {
      console.log(test)
   }
}
```

以上的代码实际上是：

```
function aa(flag) {
    var test // 变量提升，函数最顶部
    if(flag) {
        test = 'hello man';
    } else {
        console.log(test);//此处访问 test 值为 undefined
    }
     //此处访问 test 值为 undefined
}
```

所以不用关心`flag`是否为`true`或者`false`。实际上，无论如何`test`都会被创建声明。接下来`ES6`主角登场：我们通常用`let`和`const`来声明，`let`表示**变量**、`const`表示**常量**。`let`和`const`都是块级作用域。怎么理解这个块级作用域？

- 在一个函数内部
- 在一个代码块内部

> 说白了只要在**{}花括号内**的代码块即可以认为`let`和`const`的作用域。

```
function aa(flag) {
    if(flag) {
        let test = 'hello man';
    }else{
        console.log(test); //test 在此处访问不到 test is not defined
    }
}
```

`let`的作用域是在它所在当前代码块，但不会被提升到当前函数的最顶部。再来说说`const`，`const`声明的变量必须提供一个值，而且会被认为是常量，意思就是它的值被设置完成后就不能再修改了。

```
const name = 'lux';
name = 'joe'; // 再次赋值此时会报错 Uncaught TypeError: Assignment to constant variable.
```

还有，如果 `const` 的是一个对象，对象所包含的值是可以被修改的。抽象一点儿说，就是对象所指向的地址不能改变，而变量成员是可以修改的。

```
const student = { name: 'cc' };
// 没毛病
student.name = 'yy';
// 如果这样子就会报错了
student  = { name: 'yy' };
```

说说TDZ(暂时性死区)，想必你早有耳闻。

```
{
    console.log(value); // 报错
    let value = 'lala';
}
```

我们都知道，JS引擎扫描代码时，如果发现变量声明，用 `var` 声明变量时会将声明提升到函数或全局作用域的顶部。但是 `let` 或者 `const` ，会将声明关进一个小黑屋也是 `TDZ` (暂时性死区)，只有执行到变量声明这句语句时，变量才会从小黑屋被放出来，才能安全使用这个变量。

哦了，说一道面试题

```
var funcs = []
for (var i = 0; i < 10; i++) {
    funcs.push(function() {
        console.log(i)
    })
}
funcs.forEach(function(func) {
    func()
})
```

这样的面试题是大家很常见，很多同学一看就知道输出十次10。但是如果我们想依次输出0到9呢？有两种解决方法，直接看一下代码：

```
// ES5知识，我们可以利用“立即调用函数”解决这个问题
var funcs = [];
for (var i = 0; i < 10; i++) {
    funcs.push(
        (function(value) {
            return function() {
                console.log(value)
            }
        })(i)
    );
```

```
}
funcs.forEach(function(func) {
    func();
})
```

```
// 再来看看es6怎么处理的
const funcs = []
for (let i = 0; i < 10; i++) {
    funcs.push(function() {
        console.log(i)
    })
}
funcs.forEach(func => func());
```

达到相同的效果，ES6 简洁的解决方案是不是更让你心动！！！

2.字符串

先聊聊模板字符串😊 `ES6` 模板字符简直是开发者的福音啊，解决了 `ES5` 在字符串功能上的痛点。

第一个用途，基本的字符串格式化。将表达式嵌入字符串中进行拼接。用 `${}` 来界定。

```
//ES5
var name = 'lux';
console.log('hello' + name);
//es6
const name = 'lux';
console.log(`hello ${name}`);//hello lux
```

第二个用途，在 `ES5` 时我们通过反斜杠()来做多行字符串或者字符串一行行拼接。 `ES6` 反引号 (``) 直接搞定。

```
// ES5
var msg = "Hi \man!";
// ES6
const template =  `<div><span>hello world</span></div>`;
```

对于字符串 `ES6+` 当然也提供了很多厉害也很有意思的方法😊 说几个常用的。

```
// 1.includes：判断是否包含然后直接返回布尔值
const str = 'hahay';
console.log(str.includes('y')); // true

// 2.repeat: 获取字符串重复n次
const str = 'he';
console.log(str.repeat(3)); // 'hehehe'
//如果你带入小数，Math.floor(num) 来处理
// s.repeat(3.1) 或者 s.repeat(3.9) 都当做成 s.repeat(3) 来处理

// 3. startsWith 和 endsWith 判断是否以 给定文本 开始或者结束
```

```
const str =  'hello world!';
console.log(str.startsWith('hello')); // true
console.log(str.endsWith('!')); // true

// 4. padStart 和 padEnd 填充字符串，应用场景：时分秒
setInterval(() => {
    const now = new Date();
    const hours = now.getHours().toString();
    const minutes = now.getMinutes().toString();
    const seconds = now.getSeconds().toString();
    console.log(`${hours.padStart(2, 0)}:${minutes.padStart(2, 0)}:${seconds.padStart(2, 0)}`)
}, 1000)
```

关于模板字符串现在比较常出现的面试题有两道。同学们不妨写试试看？

- 模拟一个模板字符串的实现。

```
let address = '北京海淀区';
let name = 'lala';
let str = '${name}在${address}上班...';
// 模拟一个方法 myTemplate(str) 最终输出 'lala在北京海淀区上班...'
function myTemplate(str) {
    // try it
}
console.log(myTemplate(str)); // lala在北京海淀区上班...
```

- 实现标签化模板(自定义模板规则)。

```
const name = 'cc';
const gender = 'male';
const hobby = 'basketball';
// 实现tag最终输出 '姓名：**cc**，性别：**male**，爱好：**basketball**'
function tag(strings) {
    // do it
}
const str = tag`姓名：${name}，性别：${gender}，爱好：${hobby}`;
console.log(str); // '姓名：**cc**，性别：**male**，爱好：**basketball**'
```

3.函数

函数默认参数

在`ES5`我们给函数定义参数默认值是怎么样？

```
function action(num) {
    num = num || 200;
    //当传入num时，num为传入的值
    //当没传入参数时，num即有了默认值200
    return num;
}
```

但细心观察的同学们肯定会发现，`num`传入为`0`的时候就是`false`，但是我们实际的需求就是要拿到`num = 0`，此时`num = 200`明显与我们的实际想要的效果明显不一样。`ES6`为参数提供了默认值。在定义函数时便初始化了这个参数，以便在参数没有被传递进去时使用。

```
function action(num = 200) {
    console.log(num)
}
action(0) // 0
action() //200
action(300) //300
```

箭头函数

`ES6`很有意思的一部分就是函数的快捷写法。也就是箭头函数。

箭头函数最直观的三个特点。

- 不需要`function`关键字来创建函数
- 省略`return`关键字
- 继承当前上下文的`this`关键字

```
//例如：
[1,2,3].map(x => x + 1)

//等同于：
[1,2,3].map((function(x){
    return x + 1
}).bind(this))
```

说个小细节。

当你的函数**有且仅有**一个参数的时候，是可以省略掉括号的。当你函数返回**有且仅有**一个表达式的时候可以省略`{}`和`return`；例如:

```
var people = name => 'hello' + name
//参数name就没有括号
```

作为参考

```
var people = (name, age) => {
    const fullName = 'hello' + name
    return fullName
}
//如果缺少()或者{}就会报错
```

要不整一道笔试题？哈哈哈哈哈哈哈哈。我不管我先上代码了

```
// 请使用ES6重构以下代码
var calculate = function(x, y, z) {
    if (typeof x != 'number') { x = 0 }
    if (typeof y != 'number') { y = 6 }

    var dwt = x % y
    var result

    if (dwt == z) { result = true }
    if (dwt != z) { result = false }

    return result
}
```

```
const calculate = (x, y, z) => {
    x = typeof x !== 'number' ? 0 : x
    y = typeof y !== 'number' ? 6 : y
    return x % y === z
}
```

4.拓展的对象功能

对象初始化简写，`ES5`我们对于对象都是以**键值对**的形式书写，是有可能出现键值对重名的。例如：

```
function people(name, age) {
    return {
        name: name,
        age: age
    };
}
```

键值对重名，`ES6`可以简写如下：

```
function people(name, age) {
    return {
        name,
        age
    };
}
```

`ES6` 同样改进了为对象字面量方法赋值的语法。`ES5` 为对象添加方法：

```
const people = {
    name: 'lux',
    getName: function() {
        console.log(this.name);
    }
}
```

`ES6` 通过省略冒号与 `function` 关键字，将这个语法变得更简洁

```
const people = {
    name: 'lux',
    getName () {
        console.log(this.name);
    }
}
```

`ES6` 对象提供了 `Object.assign()` 这个方法来实现浅复制。 `Object.assign()` 可以把任意多个源对象自身可枚举的属性拷贝给目标对象，然后返回目标对象。第一参数即为目标对象。在实际项目中，我们为了不改变源对象。一般会把目标对象传为 `{}`

```
const objA = { name: 'cc', age: 18 }
const objB = { address: 'beijing' }
const objC = {} // 这个为目标对象
const obj = Object.assign(objC, objA, objB)

// 我们将 objA objB objC obj 分别输出看看
console.log(objA)   // { name: 'cc', age: 18 }
console.log(objB) // { address: 'beijing' }
console.log(objC) // { name: 'cc', age: 18, address: 'beijing' }
console.log(obj) // { name: 'cc', age: 18, address: 'beijing' }

// 是的，目标对象ObjC的值被改变了。
// so，如果objC也是你的一个源对象的话。请在objC前面填在一个目标对象{}
Object.assign({}, objC, objA, objB)
```

5.更方便的数据访问--解构

数组和对象是 `JS` 中最常用也是最重要表示形式。为了简化提取信息， `ES6` 新增了**解构**，这是将一个数据结构分解为更小的部分的过程

`ES5` 我们提取对象中的信息形式如下：

```
const people = {
    name: 'lux',
    age: 20
}
const name = people.name;
const age = people.age;
console.log(name + ' --- ' + age);
```

是不是觉得很熟悉，没错，在ES6之前我们就是这样获取对象信息的，一个一个获取。现在，解构能让我们从对象或者数组里取出数据存为变量，例如

```
//对象
const people = {
    name: 'lux',
    age: 20
}
const { name, age } = people;
console.log(`${name} --- ${age}`);
//数组
const color = ['red', 'blue'];
const [first, second] = color;
console.log(first); //'red'
console.log(second); //'blue'
```

要不来点儿面试题，看看自己的掌握情况？

```
// 请使用 ES6 重构一下代码

// 第一题
var jsonParse = require('body-parser').jsonParse;

// 第二题
var body = request.body;
var username = body.username;
var password = body.password;
```

```
// 1.
import { jsonParse } from 'body-parser';
// 2.
const { body, body: { username, password } } = request;
```

6.Spread Operator 展开运算符

ES6中另外一个好玩的特性就是`Spread Operator`也是三个点儿...接下来就展示一下它的用途。

组装对象或者数组

```
//数组
const color = ['red', 'yellow'];
const colorful = [...color, 'green', 'pink'];
console.log(colorful); //[red, yellow, green, pink]

//对象
const alp = { fist: 'a', second: 'b'};
const alphabets = { ...alp, third: 'c' };
console.log(alphabets); //{ "fist": "a", "second": "b", "third": "c"}
```

有时候我们想获取数组或者对象除了前几项或者除了某几项的其他项

```
//数组
const number = [1,2,3,4,5];
const [first, ...rest] = number;
console.log(rest);//2,3,4,5
//对象
const user = {
    username: 'lux',
    gender: 'female',
    age: 19,
    address: 'peking'
}
const { username, ...rest } = user;
console.log(rest); //{"address": "peking", "age": 19, "gender": "female"}
```

对于 Object 而言，还可以用于组合成新的 Object 。(ES2017 stage-2 proposal) 当然如果有重复的属性名，右边覆盖左边

```
const first = {
    a: 1,
    b: 2,
    c: 6,
}
const second = {
    c: 3,
    d: 4
}
const total = { ...first, ...second };
console.log(total); // { a: 1, b: 2, c: 3, d: 4 }
```

7.import 和 export

`import`导入模块、`export`导出模块。

```
//全部导入
import people from './example'

//有一种特殊情况，即允许你将整个模块当作单一对象进行导入
```

```
//该模块的所有导出都会作为对象的属性存在
import * as example from "./example.js"
console.log(example.name)
console.log(example.age)
console.log(example.getName())

//导入部分
import {name, age} from './example'

// 导出默认，有且只有一个默认
export default App

// 部分导出
export class App extend Component {};
```

以前有人问我，导入的时候有没有大括号的区别是什么。下面是我在工作中的总结：

```
1.当用export default people导出时，就用 import people 导入（不带大括号）

2.一个文件里，有且只能有一个export default。但可以有多个export。

3.当用export name 时，就用import { name }导入（记得带上大括号）

4.当一个文件里，既有一个export default people，又有多个export name 或者 export age时，导入就用
import people, { name, age }

5.当一个文件里出现n多个 export 导出很多模块，导入时除了一个一个导入，也可以用import * as example
```

8. Promise

> 在`promise`之前代码过多的回调或者嵌套，可读性差、耦合度高、扩展性低。通过`Promise`机制，扁平化的代码机构，大大提高了代码可读性；用同步编程的方式来编写异步代码，保存线性的代码逻辑，极大的降低了代码耦合性而提高了程序的可扩展性。

说白了就是用同步的方式去写异步代码。发起异步请求：

```
fetch('/api/todos')
    .then(res => res.json())
    .then(data => ({ data }))
    .catch(err => ({ err }));
```

今天看到一篇关于面试题的很有意思。

```
setTimeout(function() {
    console.log(1)
}, 0);
new Promise(function executor(resolve) {
    console.log(2);
    for( var i=0 ; i<10000 ; i++ ) {
        i == 9999 && resolve();
    }
    console.log(3);
}).then(function() {
    console.log(4);
});
console.log(5);
```

Excuse me？这个前端面试在搞事！

当然以上`promise`的知识点，这个只是冰山一角。需要更多地去学习了解一下。

9.Generators

生成器（`generator`）是能返回一个**迭代器**的函数。生成器函数也是一种函数，最直观的表现就是比普通的`function`多了个星号*，在其函数体内可以使用`yield`关键字,有意思的是函数会在每个`yield`后暂停。

这里生活中有一个比较形象的例子。咱们到银行办理业务时候都得向大厅的机器取一张排队号。你拿到你的排队号，机器并不会自动为你再出下一张票。也就是说取票机"暂停"住了，直到下一个人再次唤起才会继续吐票。

`OK`。说说迭代器。当你调用一个`generato`r时，它将返回一个迭代器对象。这个迭代器对象拥有一个叫做`next`的方法来帮助你重启`generator`函数并得到下一个值。`next`方法不仅返回值，它返回的对象具有两个属性：`done`和`value`。`value`是你获得的值，`done`用来表明你的`generator`是否已经停止提供值。继续用刚刚取票的例子，每张排队号就是这里的`value`，打印票的纸是否用完就这是这里的`done`。

```
// 生成器
function *createIterator() {
    yield 1;
    yield 2;
    yield 3;
}

// 生成器能像正规函数那样被调用，但会返回一个迭代器
let iterator = createIterator();

console.log(iterator.next().value); // 1
console.log(iterator.next().value); // 2
console.log(iterator.next().value); // 3
```

那生成器和迭代器又有什么用处呢？

围绕着生成器的许多兴奋点都与异步编程直接相关。异步调用对于我们来说是很困难的事，我们的函数并不会等待异步调用完再执行，你可能会想到用回调函数，（当然还有其他方案比如`Promise`比如`Async/await`）。

生成器可以让我们的代码进行等待。就不用嵌套的回调函数。使用`generator`可以确保当异步调用在我们的`generator`函数运行一下行代码之前完成时暂停函数的执行。

那么问题来了，咱们也不能手动一直调用`next()`方法，你需要一个能够调用生成器并启动迭代器的方法。就像这样子的

```
function run(taskDef) { //taskDef即一个生成器函数
    // 创建迭代器，让它在别处可用
    let task = taskDef();

    // 启动任务
    let result = task.next();

    // 递归使用函数来保持对 next() 的调用
    function step() {

        // 如果还有更多要做的
        if (!result.done) {
            result = task.next();
            step();
        }
    }

    // 开始处理过程
    step();
}
```

> 生成器与迭代器最有趣、最令人激动的方面，或许就是可创建外观清晰的异步操作代码。你不必到处使用回调函数，而是可以建立貌似同步的代码，但实际上却使用 yield 来等待异步操作结束。