这篇文章介绍了5个 ES6 特性，使你的 JavaScript 代码变的更好。不用说，我们大多数前端开发工程师非常关注 JavaScript 的性能和特性，这就是为什么 ES6 对于我们来说是如此令人兴奋。

ES6的变化是巨大的，是令人兴奋的，也有令人困惑的地方。在本文中，我将介绍5个 ES6 的新特性，您可以立即使用它们来增强你的 JavaScript代码，以及哪些特性不建议使用。

Ecma是什么？

JavaScript 多年来已经有了很多版本，但总体而言，进化很缓慢，直到最近。 我觉得在过去的两年里，发生了比过去二十年更多的变化。 ES6 是 EcmaScript 6 的简称。它也称为 EcmaScript 2015 或 ES 2015。这些都是“新的Javascript”的不同名称。

我不完全确定为什么最近 JS 改变如此之大，但似乎是因为主要的 JavaScript 引擎厂商终于有兴趣推动该语言的发展了。

此外，像 [TypeScript](https://www.typescriptlang.org/" \t "blank) 这样的 [转移器](https://scotch.io/tutorials/javascript-transpilers-what-they-are-why-we-need-them) 的出现，使得在将其用于浏览器之前，可以使用新的语言特性。这两个组合将大大推动 JavaScript 的发展。

JS很重要，因为它是 web 的构造者，并越来越多的服务器端使用开始 Node ，越来越多的人使用 Cordova，React Native 和 Electron 来开发手机和桌面应用程序。

简而言之：JS无处不在。

所以重要的是我们来推动它。不演化的语言开始走向死亡。改善语言意味着我们可以改进我们的代码。我们的应用程序可以更少地出现错误。其中一些特性能使代码更快地运行。所以让我们从变量开始，来一起来看看 ES6 的新特性。

变量的问题

在 JavaScript 中，当您要创建一个变量时，可以使用 var 关键字。 var 非常强大，但它有一些问题。首先，变量总是不断变化的。没有办法在 JavaScript 中定义一个真正的常量。您可以通过创建具有不可变属性的对象来近似伪造常量。您可以覆盖该属性的setter，使其不能由外部代码设置，但它需要很多额外的代码，需要一个完整的对象来代替一个简单的变量。

JavaScript 代码:

1. var VARS = {
2. foo
3. set = function() { }
4. get = function() { return 42 }
5. }
6. VARS.foo = 42; // now I have a constant

常规Javascript变量还有一个作用域问题。 看看这段代码。

JavaScript 代码:

1. function badCode() {
2. if(foo) {
3. var x = 5;
4. }
5. if(bar) {
6. console.log(x);
7. }
8. }

在函数体内通过 var 声明的变量，在函数体内全局可用， var 声明的变量没有块级作用域。如果你认为上面的 console.log 语句不能正常执行，因为在 bar 定义在不同的块中。那么我告诉你，在类似于 java 或 C# 等语言中，你是对的，但是在 JavaScript 中，var 声明的变量，在函数体内全局可用，而不是在语句块中可用。

当我们添加变量提升时，它会变得更糟。 考虑这个代码：

JavaScript 代码:

1. function badCode() {
2. console.log("foo is",foo);
3. var foo = 'bar';
4. }

我在 foo 定义之前使用它。 这段代码意味着什么？ JavaScript 引擎会将变量声明提升到函数块的顶部。 但变量提升很容易引发一些难以诊断的细微错误。

看看这个 for 循环：

JavaScript 代码:

1. for(var i=0; i<5; i++) {
2. console.log(i);
3. }
4. console.log(i);

该变量 i 仅在循环体内使用，但我仍然可以在循环体外部引用它。 这可能会引发大量的 bug 。

好消息是我们不用再必须使用 var 定义变量了，我们可以使用 const 和 let。

介绍 const 和 let

新的关键字 const 完全符合这个名字的意思。它可以定义一个真正的常量。如果您尝试重新设置常量值，您将收到运行时错误。更好的是，Lint语法检查工具可以在编译时检测出这种错误，所以您可以在开发时较早的发现这种错误。

另一种新的变量定义方式是使用 let 关键字。let 就像 var ，但是它是块级作用域而不是函数作用域。下面代码console.log(x)会报错：

JavaScript 代码:

1. function goodCode() {
2. if(foo) {
3. let x = 5;
4. }
5. if(bar) {
6. console.log(x); //error
7. }
8. }

让我们回顾一下 for 循环的例子：

JavaScript 代码:

1. function goodCode() {
2. for(let i=0; i<5; i++) {
3. console.log(i);
4. }
5. console.log(i); //error
6. }

现在，我的变量只限于for循环体内使用。没有办法可以不经意地使用。另外，let 定义的变量不能提升，所以所有这些不可思议的移动变量都消失了。

新的关键字 const 和 let 可以完全替代 var 。 使用现代浏览器和最新版本的Node，没有任何理由使用 var 了，甩掉历史包袱。（愚人码头注：个人认为还是要看使用环境。）

超级字符串

ES6引入了一种新类型的字符串，称为模板字面量。我更喜欢称之为超级字符串。你使用一个超级字符串就像一个常规的字符串，但是你不光可以使用单引号或双引号，你还可以使用反引号`。

JavaScript 代码:

1. var q = 'foo';
2. var qq = "foo";
4. var bq = `foo`;
6. var qq = "Sally sells \"sea shells\"";
7. var bq = `Sally sells "sea shells"`;

到现在为止还挺好，但没有什么非同寻常的地方。不过它确实有一个直接的优势。如果您需要在字符串中使用双引号或单引号，则不必再转义（愚人码头注：原来单引号双引号配对不是也可以吗？不知道作者为什么这样写。嘎嘎)。然而，超级字符串还有其他一些技巧。

多行字符串串联

终于我们可以有真正的多行字符串了。如果你需要引用几行的字符串？你不必再使用加号的方式链接字符串。直接放入换行符，它就能正常工作。

JavaScript 代码:

1. var q = 'foo';
2. var qq = "foo";
4. var bq = `foo`;
6. var qq = "Sally sells \"sea shells\"";
7. var bq = `Sally sells "sea shells"`;

表达式换码

另一个新特性是表达式换码（愚人码头注：前端说的比较多的是替换符)。在超级字符串中，您可以在${}的括号内放入任何有效的 JavaScript 表达式。这比双引号更加干净，最新的IDE一般都会语法高亮显示这些表达式。

JavaScript 代码:

1. var name = "Alice";
2. var greeting = `Good morning ${name}`;
3. var amount = 5;
4. var price = 3;
5. var sales = `That costs ${amount\*price} dollars`;

将表达式换码与多行字符串支持相结合，为我们提供了很棒的HTML模板。

JavaScript 代码:

1. var template = `
2. <div>
3. <h3>Good Morning ${name}</h3>
4. <p>
5. That item will cost you
6. <b>${amount\*price}</b>
7. dollars.
8. </p>
9. </div>`

JavaScript 代码:



箭头函数

上面说的是字符串和变量，现在让我们来看看函数。如果您以前听过ES6，很多可能都是关于箭头函数的。这是更紧凑的常规函数语法。他们也有一个非常重要的区别：this 变量指向的是不同的东西。

假设你想循环一个数组使其元素的值乘2，并且生成一个新数组。你可以用下面的 for 循环来做，但是这样会产生额外的变量，并且可以很容易写错。

JavaScript 代码:

1. var output = [];
2. for(var i=0; i<;input.length; i++) {
3. output[i] = input[i] \* 2;
4. }

JavaScript数组有一个新方法 map,它在每个元素上调用一个函数来生成一个新的元素，然后将其放入新数组中。代码如下：

JavaScript 代码:

1. var output = input.map(function(x) {
2. return x \* 2;
3. });

这看起来更好，更简洁。x \* 2 部分是惟一实际工作的部分。 其余的是语法开销。如果使用箭头功能，我们可以这样做：

JavaScript 代码:

1. var output = input.map((x)=>x\*2);

哇！这个更加简洁了。让我解释一下上面的代码。箭头函数可以重写函数，而不需要实际的function关键字。而是在包含函数参数的括号后面跟 =&gt; 符号。

JavaScript 代码:

1. //常规函数
2. function (x) {
3. return x \* 2;
4. }
5. //箭头函数样式
6. (x) => {
7. return x \* 2;
8. }

箭头函数让我们写同样的代码更加简洁。但我们还可以继续简化。删除空格，相同的更简短。

JavaScript 代码:

1. (x) => { return x \* 2; }

我们还可以使它更简短。如果箭头函数只包含一个表达式，我们可以删除return，大括号和分号，写成一个单行表达式，它会自动返回它的值。这样就简化到极致了。

JavaScript 代码:

1. (x) => x \* 2
2. var output = input.map((x)=>x\*2);

箭头函数可以使您的代码非常紧凑和强大。 但是还有一个非常重要的事情是：它修复了 this 。

this 魔咒

在 JavaScript 中，最不可思议是变量 this 总是引用函数被调用的那个对象。所以像下面的代码肯能不会是你认为的那样执行。

JavaScript 代码:

1. function App() {
2. this.clicked = false;
3. var button = document.getElementById('button');
4. button.addEventListener('click', function(evt) {
5. this.clicked = true; //不会安装你想的那样执行
6. });
7. }

当您使用其他对象时， this 上下文可能与预期的不同。当你将一个函数传递到其他地方被调用时，它可能会使用不同的 this 调用该函数。如果您将一个事件处理程序添加到 button ， this 将指向 button 。有时这就是你想要的，但在上面的代码中不是。我们希望 this 指向 App 对象，而不是 button 。

this 是JavaScript长期存在的问题。很常见的做法是，开发人员创造了一种 self 模式，使用临时变量 self 保存对 this 的正确引用。看起来很恶心，但它能正常工作。

JavaScript 代码:

1. function App() {
2. this.clicked = false;
3. var button = document.getElementById('button');
4. var self = this; //特别注意这一行
5. button.addEventListener('click', function(evt) {
6. self.clicked = true;
7. });
8. }

另一种解决问题的方法是 bind 绑定这个函数。bind 方法强制 this 执行一个特定的对象，不管函数后面如何被调用。（愚人码头注： <http://www.css88.com/archives/5611> ）

JavaScript 代码:

1. function App() {
2. this.clicked = false;
3. var button = document.getElementById('button');
4. var callback = (function(evt) {
5. this.clicked = true
6. }).bind(this);//特别注意这一行
7. button.addEventListener('click',callback);
8. }

再次执行， this 能正常指向 App 对象，但它不是很好。我们需要写额外的代码，并且 bind 会带来额外的开销。箭头函数为我们提供了更好的方法。

JavaScript 代码:

1. function App() {
2. this.clicked = false;
3. var button = document.getElementById('button');
4. button.addEventListener('click',()=>{
5. this.clicked = true;
6. });
7. }

箭头函数自动捕获该函数定义时作用域中的 this 变量，而不是来自函数被调用的作用域中。这意味着您可以将函数传递给其他地方，使用时要知道 this 正确的指向。在上面的代码中，没有任何的 hack ，就能按照我们所期望的那样执行。

简而言之，箭头函数真的很棒。我可以尽可能地使用它。它们使您的代码更短，更容易阅读，而且 this 也变得很明智。

Promises

箭头函数的另一个很强大的特性是它们能与 Promises 配合的很好。Promise 是 JavaScript 中一种新的对象，旨在帮助需要很长时间执行的事情。JavaScript 没有线程，所以如果你想做一些可能需要很长时间的事情，那么你必须使用回调。

例如，在Node中，您可能需要加载文件，解析它，做一个数据库请求，然后写一个新的文件。这些都必须按顺序完成，但它们都是异步的，所以你必须嵌套你的回调。这就产生了被JS开发人员称之为“金字塔”式的代码风格。你需要大量的嵌套代码。

JavaScript 代码:

1. fs.readFile("file.txt", function(err, file) {
2. db.fetchNames(function(err, names) {
3. processFile(file, names, function(err, outfile) {
4. fs.writeFile('file.txt',outfile, function(err, status) {
5. console.log("we are done writing the file");
6. })
7. });
8. })
9. });

这段代码很丑，很难理解，而且有很多讨厌的隐藏 bug。 Promises 可以帮助我们击败 “金字塔” 式的代码风格。

JavaScript Promise 表示一个值当前可能不可用，但是将来有值的对象。使用 then 函数可以添加回调，当最终值 ready 时，调用这个回调。

JavaScript 代码:

1. var prom = makeSomePromise();
2. //value not ready yet
3. prom.then((value)=>{
4. //do something with the value
5. })

Promises then 回调很像传统的回调，但 Promises 增加了一个额外的转变 ：他们可以链式调用。让我们重温一下上面的代码。每个函数必须按顺序调用，每个函数都取决于前一个函数的结果。使用Promises ，我们可以这样做。

JavaScript 代码:

1. fs.readFile("file.txt")
2. .then((file) => {
3. return db.fetchNames().then((names)=>{
4. return processFile(file,names)
5. })
6. })
7. .then((outfile)=>{
8. return fs.writeFile('file.txt',outfile);
9. })
10. .then(()=>{
11. console.log("we are done writing the file");
12. });

请注意看，箭头函数如何使这个代码变得漂亮干净。每个 then 回调返回一个值。这个值传递给下一个函数，所以我们所有的函数都可以很容易的链式调用。

现在你会注意到，processFile 命令需要前面两个值的结果，但是 Promises 只传递一个值。我们也不用关心readFile和fetchNames执行的顺序。我们只想知道何时完成。Promises 可以做到这一点。

Promise.all()

假设要从文件名数组中加载每个文件，并在完成时通知。我们可以用 Promise.all() 来做到这一点。all 都是 Promise 中的一个实用方法，它获取一系列 Promises ，并返回了一个新的 Promises ，当所有的子 Promises 完成时，它将得到 resolves 状态。以下是我们如何使用 Promise.all加载所有文件的示例。（假设readFile是一个读取文件返回 Promises 的函数）。

JavaScript 代码:

1. var proms = filenames.map((name)=> readFile(name));
2. Promise.all(proms).then((files)=>{
3. console.log(`we have ${files.length} files`);
4. });

现在我们可以重写我们原来的Node示例：

JavaScript 代码:

1. Promise.all([
2. fs.readFile("file.txt"),
3. db.fetchNames()
4. ])
5. .then((vals) => processFile(vals[0],vals[1]))
6. .then((outfile)=> fs.writeFile('file.txt',outfile))
7. .then(()=> console.log("we are done writing the file"));

我将读取的文件和数据库调用合并到使用Promise.all的单个 Promise 中。它将返回的值是一个数组，包含两个子 Promise 的结果，所以我可以把它们放到 processFile 中。请注意，我使用了缩写箭头语法使代码更小更干净。

处理失败

现在考虑如果这些 Promises 中有一个失败会发生什么？为了处理第一个失败，我们可以将其放入 try / catch 语句块中，但下一个 then 仍然会被调用。如果第一个失败，我们希望一切都停止。Promises有另外一个技巧：捕获回调。

在下面的新版本的代码中，如果有任何失败，它将立即跳过其余的 Promises ，跳转到结尾的 catch回调中。catch 回调中，我们还可以添加更多的子句。

JavaScript 代码:

1. Promise.all([
2. fs.readFile("file.txt"),
3. db.fetchNames()
4. ])
5. .then((vals) => processFile(vals[0],vals[1]))
6. .then((outfile)=> fs.writeFile('file.txt',outfile))
7. .then(()=> console.log("we are done writing the file"))
8. .catch((e) => {
9. console.log("some error happened");
10. });

编写自定义Promises

当然 Promises 只有在我们调用的API 实际使用 Promises 的情况下才能工作。我们可以期待许多库开始转换为 Promises，当然我们也可以编写自己的 Promises 。我们使用 Promise 构造函数来实现。

JavaScript 代码:

1. function makePromise(foo,bar) {
2. return new Promise((resolve, reject) => {
3. try {
4. //do long stuff
5. resolve(value);
6. } catch {
7. reject(error);
8. }
9. });
10. }

它需要两个值：resolve 和 reject。这些都是回调函数。回调里面你可以做任何需要花很长时间执行的事情，即使是它涉及多个回调。当完全完成后，调用具有最终值的 resolve()。然后，这将发送到任何你使用 Promises 的第一个 then 子句。

如果发生错误，并且您想 reject 该值，而不是抛出错误，请使用 reject()，传递所需的任何替代值。

这是一个现实中的例子。 我一直使用 AJAX 调用，但它们可能非常丑陋，像这样。

JavaScript 代码:

1. var url = "http://api.silly.io/api/list/e3da7b3b-976d-4de1-a743-e0124ce973b8?format=json";
3. var xml = new XMLHttpRequest();
4. xml.addEventListener('load', function() {
5. var result = JSON.parse(this.responseText);
6. console.log(result);
7. });
8. xml.addEventListener('error',function(error) {
9. console.log("something bad happened");
10. });
11. xml.open("GET",url);
12. xml.send();

让我们把这个代码包装成一个承诺 Promise。

JavaScript 代码:

1. function doGet(url) {
2. return new Promise((resolve,rej)=>{
3. var xml = new XMLHttpRequest();
4. xml.addEventListener('load', function() {
5. var result = JSON.parse(this.responseText);
6. resolve(result);
7. });
8. xml.addEventListener('error',function(error) {
9. reject(error);
10. });
11. xml.open("GET",url);
12. xml.send();
13. });
14. }

基本上是相同的代码，但我可以像这样调用它。

JavaScript 代码:

1. var url = "someapi.com/dostuff?foo=bar";
2. doGet(url).then((obj)=>{
3. console.log(obj);
4. });

哇，这样更干净。实际上，我们不需要自己编写 AJAX Promise 包装器，因为有一个新的Web标准 fetch 已经为我做了这些事情。但是现在所有浏览器都还没支持 fetch ，所以我们可以使用我们自己编写的 Promise ，直到浏览器支持 fetch 的时候。

第一次包装 Promise 可能有点困难，但一旦你开始使用它们，我想你会很喜欢他们。它们可以非常容易的将多个函数集成到一个具有逻辑意义的单个工作流中，并正确地捕获所有错误。

Arrays

最后我想向大家展示一些新的 Array 功能。大多数功能对于ES6 来说不能算是新的，其实有些是相当老了。然而，它们终于得到了各方面的支持，并与 箭头函数和 Promise 结合的很好。所以我认为他们是新功能。

假设你想对数组的每个元素做一些事情。可以使用 forEach 或 map 函数代替 for 循环。

JavaScript 代码:

1. var values = [1,2,3,4,5,6,7,8,9,10];
2. values.forEach((x)=>console.log(x));
3. var doubled = values.map((x) => x\*2);

forEach函数在数组中的每个元素上运行回调。map函数也在每个元素上运行，但它将每个回调的结果存储到一个新的数组中。

如果要从数组中过滤出某些值，请使用filter函数。

JavaScript 代码:

1. //查找匹配 filter 所有的值
2. var evens = values.filter((x)=>x%2==0);

Array还具有基于某些标准在数组中查找单个值的功能。

JavaScript 代码:

1. //查找第一个匹配的值
2. var six = values.find((x) => x >= 6);
3. //找到第一个匹配的索引
4. var index = values.findIndex((x) => x >= 6);
5. //如果至少有一项匹配，则为true
6. var any = values.some((x) => x > 10);

最后，可以使用 reduce 函数将数组减少到单个值。reduce 非常强大，可以用来做很多事情，比如一个数组求和或 flatten 嵌套数组（愚人码头注：降低数组嵌套，例如将[1,[2,3],4]转换为[1,2,3,4]，这就叫 flatten）。

JavaScript 代码:

1. //将数组减少到单个值
2. var sum = values.reduce((a,b) => a+b, 0);
3. //flatten 嵌套数组
4. var list1 = [[0, 1], [2, 3], [4, 5]];
5. var list2 = [0, [1, [2, [3, [4, [5]]]]]];
6. const flatten = arr => arr.reduce(
7. (acc, val) => acc.concat(
8. Array.isArray(val) ? flatten(val) : val
9. ),
10. []
11. );
12. flatten(list1); // returns [0, 1, 2, 3, 4, 5]
13. flatten(list2); // returns [0, 1, 2, 3, 4, 5]

循环对象中的属性，您可以使用 Object.keys 获取一个包含所有属性名称数组，然后用 forEach 循环它

JavaScript 代码:

1. var obj = {
2. first:'Alice',
3. middle:'N',
4. last:'Wonderland',
5. age:8,
6. height:45,
7. }
8. Object.keys(obj).forEach((key)=>{
9. console.log("key = ", key);
10. console.log("value = ", obj[key]);
11. });

要避免的事情

我已经介绍了 ES6 今天就可以使用的五个功能，但ES6中还有许多其他功能，这个阶段你应该避免？或者是因为它们没有提供有价值的东西，或者还没有得到很好的支持。这些包括：

* Destructuring
* Modules
* Default Parameters
* Spread Operator
* Symbols
* Generators and Iterators
* Weak Maps and Weak Sets
* Proxies

解构允许你通过名称从对象中引值。它在几种情况下有用，但是我发现最好的用法是从模块中提取函数。不幸的是模块仍然是一团糟而且不要在任何地方使用，所以现在应该避免使用他们。

除了解构，建议你不要使用默认参数，和扩展操作符（...）。我发现这些比他们使用价值更麻烦，至少现在是。

Symbols，Generators（生成器），Iterators（迭代器），Weak Maps (弱映射)和 Weak Sets(弱集合)，Proxies(代理)是非常有用的，但是它们在任何地方都不支持，所以我建议你等一会儿再能使用它们。

还有一个新的类语法。它仍然使用JavaScript的原型继承，但它使得定义一个类的语法更加清晰和一致。然而，如果没有新的模块支持，它也没有价值，所以我建议等一会儿。

JavaScript 代码:

1. class Foo extends Bar {
2. constructor(stuff) {
3. super(stuff);
4. this.first = "first name"
5. }
6. doStuff() {
7. }
8. doMoreStuff() {
9. }
10. }

我可以用它吗？

大多数桌面和移动端的浏览器都支持我向你展示的所有内容。但是，根据您的用户情况，您可能需要支持旧的浏览器/旧版移动操作系统。每当你想知道他们的支持情况，去 [caniuse.com](http://caniuse.com/)。它会告诉你每个浏览器的什么版本支持什么。

如果您需要支持IE 10 以下的浏览器，那么可以使用像 [TypScript](https://www.typescriptlang.org/" \t "_blank) 或 [Babel](https://babeljs.io/) 这样的转换器。

所以这些ES6的五个很棒的功能，你可以立即开始使用。

示例代码

我们创建了一个使用 箭头函数 和 Promises 的 [示例代码的目录](https://www.pubnub.com/blocks-catalog/?utm_source=Inbound&utm_medium=scotchio&utm_campaign=IN-CY17-Q1-XXX-XXX-March-14&utm_term=utm-generator&utm_content=utm-example/) ，还有许多实用的Web服务进行通信，如实时语言翻译，地理编码和chatbot apis等等。

原文：[5 Features You Can Use in ES6 Today](https://scotch.io/bar-talk/five-things-you-can-use-in-es6-today)