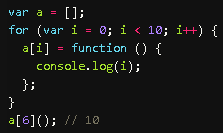
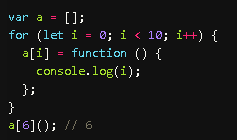
1. **let和const命令**
2. let类似于var，用于声明变量，但let声明的变量只在let所在的代码块内有效。
3. let和var在for循环中的区别：

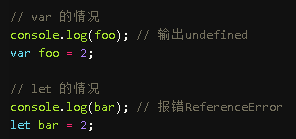


var的i作用于全局，所以全局仅有一个i变量。所以最后i的值也是来自于最后一次的循环赋值。

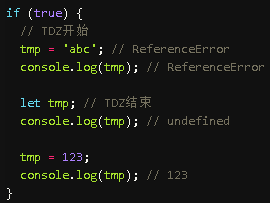


let的i只在本轮循环有效，所以每一次循环的i其实都是一个新的变量。JavaScript引擎内部会记住上一轮循环的值，初始化本轮的变量i时，就在上一轮的循环基础上进行计算。

1. for循环中，设置循环变量的那部分是一个父作用域，循环体内部是一个单独的子作用域
2. 和var不同，let不存在变量提升。即变量没有被let声明之前，使用这个变量都会报错。



1. 暂时性死区：如果区块内存在let或const命令，这个区块对这些命令声明的变量，从一开始就形成了封闭作用域。若在声明之前使用这些变量就会报错。

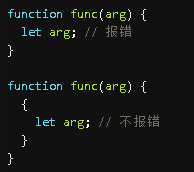


这意味着，typeof运算符不再百分之百安全。



本质：只要一进入当前作用域，所要使用的变量就已经存在了，但是不可获取，只有等到声明变量的那一行代码出现，才可以获取和使用该变量。

1. let不允许在相同作用域内，重复声明同一个变量。因此不能再函数内部重新声明函数。



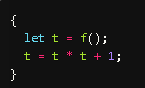
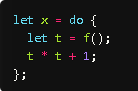
1. 块级作用域能够解决ES5中只有全局作用域和函数作用域带来的不合理场景。
2. 内层变量可能会覆盖外层变量
3. 用来计数的循环变量泄露为全局变量

而块级作用域：

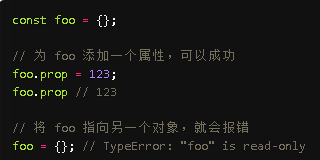
1. 块级作用域可以相互嵌套
2. 外层作用域无法读取内层作用域的变量
3. 内层作用域可以定义外层作用域的同名变量
4. 让立即执行函数表达式（IIFE），即闭包的写法不再必要。
5. 块级作用域与函数声明：
6. 考虑到环境导致的行为差异太大，应避免在块级作用域内声明函数，如果确实必要，也应该写成函数表达式而不是函数声明语句。



1. ES6的块级作用域允许声明函数的规则只在使用大括号的情况下成立。
2. do表达式：在左图中，我们知道块级作用域没有返回值，除非t是全局变量。在右图中，我们在块级作用域之前加上do，使他变为do表达式，就能得到整个块级作用域的返回值。

1. const命令：
2. 声明一个只读的常量，声明的时候必须立即赋值且不能再更改。
3. 只在声明所在块级作用域内有效。
4. 声明的常量不提升，同样存在暂时性死区。
5. 声明的常量不可重复声明。
6. 本质：const保证的不是变量的值不得改动，而是变量指向的那个内存地址不得改动。对于简单类型的数据（数值、字符串、布尔值），值就保存在变量指向的那个内存地址，因此等同于常量。但对于复合类型的数据（主要是对象和数组），变量指向的内存地址，保存的只是一个指针，const只能保证这个指针是固定的，至于它指向的数据结构是不是可变的，就完全不能控制了。所以对于复合数据类似会有如下情况：



可以将对象冻结，或者将对象的属性也冻结：

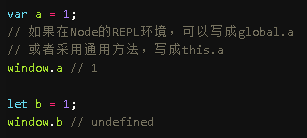


1. **ES6中六种声明变量方法**

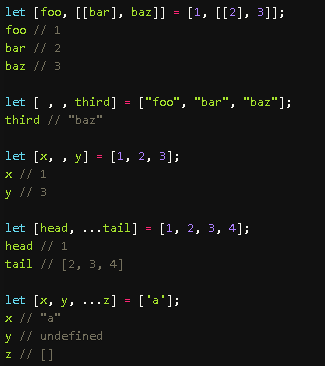
var function let const import class

1. **顶层对象的属性**

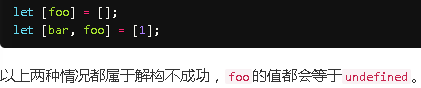
在ES6中，除了var和function声明的全局变量依然是顶层对象的属性，let、const、class声明的全局变量则不再属于顶层对象的属性。



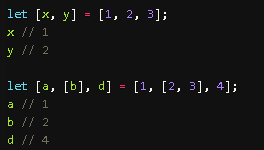
1. **变量的解构赋值**
2. 数组的解构赋值
3. 正常解构：



1. 不解构：



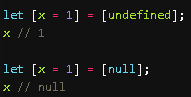
1. 不完全解构：



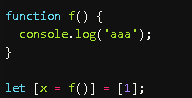
1. 如果等号右边不是数组（严格说是不可遍历的结构），就会报错
2. 对于Set结构也能使用数组的结构赋值：



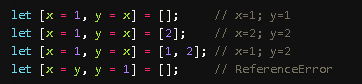
1. 只要某种数据结构有Iterator接口，就能采用数组形式的解构赋值。
2. 默认值
3. 解构赋值允许指定默认值，但对应的数组成员必须严格等于underfined，否则默认值不生效。



1. 如果默认值是一个表达式，那这个表达式是惰性求值，下面代码只有在x取不到值的时候才会触发f（）去求值。



1. 默认值可以引用解构赋值的其他变量，但该变量必须已经声明。



1. 对象的解构赋值

重点就是要分清楚匹配模式和变量。