# 1. let、const 和 block 作用域

let 允许创建块级作用域，ES6 推荐在函数中使用 let 定义变量，而非 var：

```
var a = 2;
{
  let a = 3;
  console.log(a); // 3
}
console.log(a); // 2
```

同样在块级作用域有效的另一个变量声明方式是 const，它可以声明一个常量。ES6 中，const 声明的常量类似于指针，它指向某个引用，也就是说这个「常量」并非一成不变的，如：

```
{
  const ARR = [5,6];
  ARR.push(7);
  console.log(ARR); // [5,6,7]
  ARR = 10; // TypeError
}
```

有几个点需要注意：

- let 关键词声明的变量不具备变量提升（hoisting）特性
- let 和 const 声明只在最靠近的一个块中（花括号内）有效
- 当使用常量 const 声明时，请使用大写变量，如：CAPITAL_CASING
- const 在声明时必须被赋值

# 2. 箭头函数（Arrow Functions）

ES6 中，箭头函数就是函数的一种简写形式，使用括号包裹参数，跟随一个 =>，紧接着是函数体：

```
var getPrice = function() {
  return 4.55;
};

// Implementation with Arrow Function
var getPrice = () => 4.55;
```

需要注意的是，上面例子中的 getPrice 箭头函数采用了简洁函数体，它不需要 return 语句，下面这个例子使用的是正常函数体：

```
let arr = ['apple', 'banana', 'orange'];

let breakfast = arr.map(fruit => {
  return fruit + 's';
});

console.log(breakfast); // apples bananas oranges
```

当然，箭头函数不仅仅是让代码变得简洁，函数中 this 总是绑定总是指向对象自身。具体可以看看下面几个例子：

```
function Person() {
  this.age = 0;

  setInterval(function growUp() {
    // 在非严格模式下，growUp() 函数的 this 指向 window 对象
    this.age++;
  }, 1000);
}
var person = new Person();
```

我们经常需要使用一个变量来保存 this，然后在 growUp 函数中引用：

```
function Person() {
  var self = this;
  self.age = 0;

  setInterval(function growUp() {
    self.age++;
  }, 1000);
}
```

而使用箭头函数可以省却这个麻烦：

```
function Person(){
  this.age = 0;

  setInterval(() => {
    // |this| 指向 person 对象
    this.age++;
  }, 1000);
}

var person = new Person();
```

## 3. 函数参数默认值

ES6 中允许你对函数参数设置默认值：

```
let getFinalPrice = (price, tax=0.7) => price + price * tax;
getFinalPrice(500); // 850
```

## 4. Spread / Rest 操作符

Spread / Rest 操作符指的是 ...，具体是 Spread 还是 Rest 需要看上下文语境。

当被用于迭代器中时，它是一个 Spread 操作符：

```
function foo(x,y,z) {
  console.log(x,y,z);
}
```

```
let arr = [1,2,3];
foo(...arr); // 1 2 3
```

当被用于函数传参时，是一个 Rest 操作符：

```
function foo(...args) {
  console.log(args);
}
foo( 1, 2, 3, 4, 5); // [1, 2, 3, 4, 5]
```

## 5. 对象词法扩展

ES6 允许声明在对象字面量时使用简写语法，来初始化属性变量和函数的定义方法，并且允许在对象属性中进行计算操作：

```
function getCar(make, model, value) {
  return {
    // 简写变量
    make,  // 等同于 make: make
    model, // 等同于 model: model
    value, // 等同于 value: value

    // 属性可以使用表达式计算值
    ['make' + make]: true,

    // 忽略 `function` 关键词简写对象函数
    depreciate() {
      this.value -= 2500;
    }
  };
}

let car = getCar('Barret', 'Lee', 40000);

// output: {
//     make: 'Barret',
//     model:'Lee',
//     value: 40000,
//     makeKia: true,
//     depreciate: function()
// }
```

## 6. 二进制和八进制字面量

ES6 支持二进制和八进制的字面量，通过在数字前面添加 0o 或者0O 即可将其转换为八进制值：

```
let oValue = 0o10;
console.log(oValue); // 8
```

```
let bValue = 0b10; // 二进制使用 `0b` 或者 `0B`
console.log(bValue); // 2
```

## 7. 对象和数组解构

解构可以避免在对象赋值时产生中间变量：

```
function foo() {
  return [1,2,3];
}
let arr = foo(); // [1,2,3]

let [a, b, c] = foo();
console.log(a, b, c); // 1 2 3

function bar() {
  return {
    x: 4,
    y: 5,
    z: 6
  };
}
let {x: x, y: y, z: z} = bar();
console.log(x, y, z); // 4 5 6
```

## 8. 对象超类

ES6 允许在对象中使用 super 方法：

```
var parent = {
  foo() {
    console.log("Hello from the Parent");
  }
}

var child = {
  foo() {
    super.foo();
    console.log("Hello from the Child");
  }
}

Object.setPrototypeOf(child, parent);
child.foo(); // Hello from the Parent
             // Hello from the Child
```

## 9. 模板语法和分隔符

ES6 中有一种十分简洁的方法组装一堆字符串和变量。

- ${ ... } 用来渲染一个变量
- ` 作为分隔符

```
let user = 'Barret';
console.log(`Hi ${user}!`); // Hi Barret!
```

## 10. for...of VS for...in

for...of 用于遍历一个迭代器，如数组：

```
let nicknames = ['di', 'boo', 'punkeye'];
nicknames.size = 3;
for (let nickname of nicknames) {
  console.log(nickname);
}
// 结果: di, boo, punkeye
```

for...in 用来遍历对象中的属性：

```
let nicknames = ['di', 'boo', 'punkeye'];
nicknames.size = 3;
for (let nickname in nicknames) {
  console.log(nickname);
}
Result: 0, 1, 2, size
```

## 11. Map 和 WeakMap

ES6 中两种新的数据结构集：Map 和 WeakMap。事实上每个对象都可以看作是一个 Map。

一个对象由多个 key-val 对构成，在 Map 中，任何类型都可以作为对象的 key，如：

```
var myMap = new Map();

var keyString = "a string",
    keyObj = {},
    keyFunc = function () {};

// 设置值
myMap.set(keyString, "value 与 'a string' 关联");
myMap.set(keyObj, "value 与 keyObj 关联");
myMap.set(keyFunc, "value 与 keyFunc 关联");

myMap.size; // 3

// 获取值
myMap.get(keyString);    // "value 与 'a string' 关联"
myMap.get(keyObj);       // "value 与 keyObj 关联"
myMap.get(keyFunc);      // "value 与 keyFunc 关联"
```

**WeakMap**

WeakMap 就是一个 Map，只不过它的所有 key 都是弱引用，意思就是 WeakMap 中的东西垃圾回收时不考虑，使用它不用担心内存泄漏问题。

另一个需要注意的点是，WeakMap 的所有 key 必须是对象。它只有四个方法 delete(key),has(key),get(key) 和set(key, val)：

```
let w = new WeakMap();
w.set('a', 'b');
// Uncaught TypeError: Invalid value used as weak map key

var o1 = {},
    o2 = function(){},
    o3 = window;

w.set(o1, 37);
w.set(o2, "azerty");
w.set(o3, undefined);

w.get(o3); // undefined, because that is the set value

w.has(o1); // true
w.delete(o1);
w.has(o1); // false
```

## 12. Set 和 WeakSet

Set 对象是一组不重复的值，重复的值将被忽略，值类型可以是原始类型和引用类型：

```
let mySet = new Set([1, 1, 2, 2, 3, 3]);
mySet.size; // 3
mySet.has(1); // true
mySet.add('strings');
mySet.add({ a: 1, b:2 });
```

可以通过 forEach 和 for...of 来遍历 Set 对象：

```
mySet.forEach((item) => {
  console.log(item);
    // 1
    // 2
    // 3
    // 'strings'
    // Object { a: 1, b: 2 }
});

for (let value of mySet) {
  console.log(value);
    // 1
```

```
    // 2
    // 3
    // 'strings'
    // Object { a: 1, b: 2 }
}
```

Set 同样有 delete() 和 clear() 方法。

**WeakSet**

类似于 WeakMap，WeakSet 对象可以让你在一个集合中保存对象的弱引用，在 WeakSet 中的对象只允许出现一次：

```
var ws = new WeakSet();
var obj = {};
var foo = {};

ws.add(window);
ws.add(obj);

ws.has(window); // true
ws.has(foo);    // false, foo 没有添加成功

ws.delete(window); // 从结合中删除 window 对象
ws.has(window);    // false, window 对象已经被删除
```

## 13. 类

ES6 中有 class 语法。值得注意是，这里的 class 不是新的对象继承模型，它只是原型链的语法糖表现形式。

函数中使用 static 关键词定义构造函数的的方法和属性：

```
class Task {
  constructor() {
    console.log("task instantiated!");
  }

  showId() {
    console.log(23);
  }

  static loadAll() {
    console.log("Loading all tasks..");
  }
}

console.log(typeof Task); // function
let task = new Task(); // "task instantiated!"
```

```
task.showId(); // 23
Task.loadAll(); // "Loading all tasks.."
```

类中的继承和超集：

```
class Car {
  constructor() {
    console.log("Creating a new car");
  }
}

class Porsche extends Car {
  constructor() {
    super();
    console.log("Creating Porsche");
  }
}

let c = new Porsche();
// Creating a new car
// Creating Porsche
```

extends 允许一个子类继承父类，需要注意的是，子类的constructor 函数中需要执行 super() 函数。

当然，你也可以在子类方法中调用父类的方法，如super.parentMethodName()。

在 这里 阅读更多关于类的介绍。

有几点值得注意的是：

- 类的声明不会提升（hoisting），如果你要使用某个 Class，那你必须在使用之前定义它，否则会抛出一个 ReferenceError 的错误
- 在类中定义函数不需要使用 function 关键词

## 14. Symbol

Symbol 是一种新的数据类型，它的值是唯一的，不可变的。ES6 中提出 symbol 的目的是为了生成一个唯一的标识符，不过你访问不到这个标识符：

```
var sym = Symbol( "some optional description" );
console.log(typeof sym); // symbol
```

注意，这里 Symbol 前面不能使用 new 操作符。

如果它被用作一个对象的属性，那么这个属性会是不可枚举的：

```
var o = {
    val: 10,
    [ Symbol("random") ]: "I'm a symbol",
};

console.log(Object.getOwnPropertyNames(o)); // val
```

如果要获取对象 symbol 属性，需要使用Object.getOwnPropertySymbols(o)。

## 15. 迭代器（Iterators）

迭代器允许每次访问数据集合的一个元素，当指针指向数据集合最后一个元素是，迭代器便会退出。它提供了 next() 函数来遍历一个序列，这个方法返回一个包含 done 和 value 属性的对象。

ES6 中可以通过 Symbol.iterator 给对象设置默认的遍历器，无论什么时候对象需要被遍历，执行它的 @@iterator 方法便可以返回一个用于获取值的迭代器。

数组默认就是一个迭代器：

```
var arr = [11,12,13];
var itr = arr[Symbol.iterator]();

itr.next(); // { value: 11, done: false }
itr.next(); // { value: 12, done: false }
itr.next(); // { value: 13, done: false }

itr.next(); // { value: undefined, done: true }
```

你可以通过 [Symbol.iterator]() 自定义一个对象的迭代器。

## 16. Generators

Generator 函数是 ES6 的新特性，它允许一个函数返回的可遍历对象生成多个值。

在使用中你会看到 * 语法和一个新的关键词 yield:

```
function *infiniteNumbers() {
  var n = 1;
  while (true){
    yield n++;
  }
}

var numbers = infiniteNumbers(); // returns an iterable object

numbers.next(); // { value: 1, done: false }
numbers.next(); // { value: 2, done: false }
numbers.next(); // { value: 3, done: false }
```

每次执行 yield 时，返回的值变为迭代器的下一个值。

## 17. Promises

ES6 对 Promise 有了原生的支持，一个 Promise 是一个等待被异步执行的对象，当它执行完成后，其状态会变成 resolved 或者rejected。

```
var p = new Promise(function(resolve, reject) {
  if (/* condition */) {
    // fulfilled successfully
    resolve(/* value */);
```

```
  } else {
    // error, rejected
    reject(/* reason */);
  }
});
```

每一个 Promise 都有一个 .then 方法，这个方法接受两个参数，第一个是处理 resolved 状态的回调，一个是处理 rejected 状态的回调：

```
p.then((val) => console.log("Promise Resolved", val),
       (err) => console.log("Promise Rejected", err));
```