ES6 语法

let 命令

ES6 新增了 let 命令，用来声明变量。它的用法类似于 var，但是所声明的变量，只在 let 命令所在的代码块内有效。

```
{
    let name = "Song";
    var email = "1746948032@qq.com";
}
console.log(name);
console.log(email);
```

不能获取块内作用域name的值

```
1746948032@qq.com
undefined
```

for 循环的一个特别之处，就是循环语句部分是一个父作用域，而循环体内部是一个单独的子作用域。

```
for(let i = 0; i < 5; i++){
    let i = "song";
    console.log(i);
}
song
undefined
```

不存在变量提升

var 命令会发生”变量提升“现象，即变量可以在声明之前使用，值为 undefined。

为了纠正这种现象，let 命令改变了语法行为，它所声明的变量一定要在声明后使用，否则报错。

```
console.log(foo);
var foo = 2;
console.log(bar);
let bar = 2;
undefined
Uncaught ReferenceError: bar is not defined
    at <anonymous>:3:13
```

暂时性死区

只要块级作用域内存在 let 命令，它所声明的变量就“绑定”（binding）这个区域，不再受外部的影响。

```
var tmp = '123';
if(true){
    tmp = 'abc';
    let tmp;
}
```

块级作用域内只允许一个tmp；不接受外部定义；必须先定义后使用！

```
Uncaught ReferenceError: tmp is not defined
    at <anonymous>:3:6
```

ES6 明确规定，如果区块中存在 let 和 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。凡是在声明之前就使用这些变量，就会报错。

总之，在代码块内，使用 let 命令声明变量之前，该变量都是不可用的。这在语法上，称为“暂时性死区”（temporal dead zone，简称 TDZ）。
这样的设计是为了让大家养成良好的编程习惯，变量一定要在声明之后使用，否则就报错。
暂时性死区的本质就是，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。

块级作用域
为什么需要块级作用域?
ES5 只有全局作用域和函数作用域，没有块级作用域，这带来很多不合理的场景。
第一种场景，内层变量可能会覆盖外层变量。

```
> var tmp = new Date();
  function f(){
      console.log(tmp);
      if(false){
          var tmp = "hello world";
      }
  }
  f();
  undefined
< undefined
```

ES5的变量提升，导致内层的tmp变量覆盖了外层的tmp变量

```
> var tmp = new Date();
  function f(){
      console.log(tmp);
      if(false){
          let tmp = "hello world";
      }
  }
  f();
  Mon Mar 13 2017 16:09:01 GMT+0800 (中国标准时间)
< undefined
>
```

ES6的let语法定义了块级作用域

第二种场景，用来计数的循环变量泄露为全局变量。

```
> var s = "hello";
  for(var i = 0; i < s.length; i++){
      console.log(s[i]);
  }
  console.log(i);
  h
  e
  l
  l
  o
  5
< undefined
>
```

控制循环的临时变量i泄漏成了全局变量

const 命令
const 声明一个只读的常量。一旦声明，常量的值就不能改变。

本质

const 实际上保证的并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。

ES6 声明变量的六种方法

ES5 只有两种声明变量的方法：var 命令和 function 命令。

ES6 有 var、function、let、const、import、class。

变量的解构赋值

ES6 允许按照一定模式，从数组和对象中提取值，对变量进行赋值，这被称为解构（Destructuring）。

数组的解构赋值

以前，为变量赋值，只能直接指定值。

let a = 1;

let b = 2;

let c = 3;

ES6 允许写成下面这样。

let [a, b, c] = [1, 2, 3];

上面代码表示，可以从数组中提取值，按照对应位置，对变量赋值。

对象的解构赋值

对象的解构与数组有一个重要的不同。数组的元素是按次序排列的，变量的取值由它的位置决定；而对象的属性没有次序，变量必须与属性同名，才能取到正确的值。

字符串的解构赋值

数值和布尔值的解构赋值
解构赋值时，如果等号右边是数值和布尔值，则会先转为对象。

```
let {toString: s} = 123;
s === Number.prototype.toString // true

let {toString: s} = true;
s === Boolean.prototype.toString // true
```

上面代码中，数值和布尔值的包装对象都有 toString 属性，因此变量 s 都能取到值。

解构赋值的规则是，只要等号右边的值不是对象或数组，就先将其转为对象。由于 undefined 和 null 无法转为对象，所以对它们进行解构赋值，都会报错。

```
let { prop: x } = undefined; // TypeError
let { prop: y } = null; // TypeError
```

用途
变量的解构赋值用途很多。

（1）交换变量的值

```
let x = 1;
let y = 2;

[x, y] = [y, x];
```

上面代码交换变量 x 和 y 的值，这样的写法不仅简洁，而且易读，语义非常清晰。

（2）从函数返回多个值

函数只能返回一个值，如果要返回多个值，只能将它们放在数组或对象里返回。有了解构赋

值，取出这些值就非常方便。

```
// 返回一个数组

function example() {
  return [1, 2, 3];
}
let [a, b, c] = example();

// 返回一个对象

function example() {
  return {
    foo: 1,
    bar: 2
  };
}
let { foo, bar } = example();
```

（3）函数参数的定义

解构赋值可以方便地将一组参数与变量名对应起来。

```
// 参数是一组有次序的值
function f([x, y, z]) { ... }
f([1, 2, 3]);

// 参数是一组无次序的值
function f({x, y, z}) { ... }
f({z: 3, y: 2, x: 1});
```

（4）提取 JSON 数据

解构赋值对提取 JSON 对象中的数据，尤其有用。

```
let jsonData = {
  id: 42,
  status: "OK",
  data: [867, 5309]
};

let { id, status, data: number } = jsonData;

console.log(id, status, number);
// 42, "OK", [867, 5309]
```

上面代码可以快速提取 JSON 数据的值。

（5）函数参数的默认值

```
jQuery.ajax = function (url, {
  async = true,
  beforeSend = function () {},
  cache = true,
  complete = function () {},
  crossDomain = false,
  global = true,
  // ... more config
}) {
  // ... do stuff
};
```

指定参数的默认值，就避免了在函数体内部再写 var foo = config.foo || 'default foo';这样的语句。

（6）遍历 Map 结构

任何部署了 Iterator 接口的对象，都可以用 for...of 循环遍历。Map 结构原生支持 Iterator 接口，配合变量的解构赋值，获取键名和键值就非常方便。

```
var map = new Map();
map.set('first', 'hello');
map.set('second', 'world');

for (let [key, value] of map) {
  console.log(key + " is " + value);
}
// first is hello
// second is world
```

如果只想获取键名，或者只想获取键值，可以写成下面这样。

```
// 获取键名
for (let [key] of map) {
  // ...
}

// 获取键值
for (let [,value] of map) {
  // ...
}
```

（7）输入模块的指定方法

加载模块时，往往需要指定输入哪些方法。解构赋值使得输入语句非常清晰。

const { SourceMapConsumer, SourceNode } = require("source-map");

```
import React, { Component } from 'react';
import { AppRegistry, Text } from 'react-native';

class HelloWorldApp extends Component {
  render() {
    return (
      <Text>Hello world!</Text>
    );
  }
}

// 注意，这里用引号括起来的'HelloWorldApp'必须和你init创建的项目名一致
AppRegistry.registerComponent('HelloWorldApp', () => HelloWorldApp);
```

函数的 name 属性
函数的 name 属性，返回该函数的函数名。

```
> function foo(){};
  console.log(foo.name);
  foo
< undefined
> (new Function).name
< "anonymous"
>
```

箭头函数
ES6 允许使用“箭头”（=>）定义函数。

var f = v => v;
上面的箭头函数等同于：

```
var f = function(v) {
  return v;
};
```

如果箭头函数的代码块部分多于一条语句，就要使用大括号将它们括起来，并且使用 return 语句返回。

var sum = (num1, num2) => { return num1 + num2; }
箭头函数使得表达更加简洁。
对象的简洁表示

```
let dessert = 'cake', drink = 'beer';
let food = {
      dessert,
      drink,
      breakfast() {
```

```
            return `今天 Song 的早餐是 ${dessert} 和 ${drink}`
        }
};
console.log(food.breakfast());
```

对象的属性和方法都可以简写。

对象的属性名表达式

```
let drink = 'hot drink';
food[drink] = 'coffee';
```

用表达式作为属性名

```
obj['a' + 'bc'] = 123;
```

对象的比较

ES5 比较两个值是否相等，只有两个运算符：相等运算符（==）和严格相等运算符（===）。它们都有缺点，前者会自动转换数据类型，后者的 NaN 不等于自身，以及+0 等于-0。JavaScript 缺乏一种运算，在所有环境中，只要两个值是一样的，它们就应该相等。

ES6 提出“Same-value equality”（同值相等）算法，用来解决这个问题。Object.is 就是部署这个算法的新方法。它用来比较两个值是否严格相等，与严格比较运算符（===）的行为基本一致。

```
Object.is('foo', 'foo')
// true
Object.is({}, {})
// false
```

不同之处只有两个：一是+0 不等于-0，二是 NaN 等于自身。

```
+0 === -0 //true
NaN === NaN // false

Object.is(+0, -0) // false
Object.is(NaN, NaN) // true
```

对象的合并与赋值

Object.assign 方法用于对象的合并，将源对象（source）的所有可枚举属性，复制到目标对象（target）。

```
Object.assign(target, source1, source2);
let breakfast = {};
Object.assign(breakfast, {drink: ''beer''});
```

读取与设置对象的 prototype(原型对象)

Object.setPrototypeOf()（写操作）、Object.getPrototypeOf()（读操作）

__proto__属性

__proto__属性（前后各两个下划线），用来读取或设置当前对象的 prototype 对象。目前，所有浏览器（包括 IE11）都部署了这个属性。

Set

ES6 提供了新的数据结构 Set。它类似于数组，但是成员的值都是唯一的，没有重复的值。
Set 本身是一个构造函数，用来生成 Set 数据结构。

```
const s = new Set();

[2, 3, 5, 4, 5, 2, 2].forEach(x => s.add(x));

for (let i of s) {
  console.log(i);
}
// 2 3 5 4
```

可以用于去除数组重复成员
Set 结构的实例有以下属性。

Set.prototype.constructor：构造函数，默认就是 Set 函数。
Set.prototype.size：返回 Set 实例的成员总数。
Set 实例的方法分为两大类：操作方法（用于操作数据）和遍历方法（用于遍历成员）。下面先介绍四个操作方法。

add(value)：添加某个值，返回 Set 结构本身。
delete(value)：删除某个值，返回一个布尔值，表示删除是否成功。
has(value)：返回一个布尔值，表示该值是否为 Set 的成员。
clear()：清除所有成员，没有返回值。

Promise
Promise 是异步编程的一种解决方案，比传统的解决方案——回调函数和事件——更合理和更强大。ES6 原生提供了 Promise 对象。
所谓 Promise，简单说就是一个容器，里面保存着某个未来才会结束的事件（通常是一个异步操作）的结果。从语法上说，Promise 是一个对象，从它可以获取异步操作的消息。Promise 提供统一的 API，各种异步操作都可以用同样的方法进行处理。
Promise 对象代表一个异步操作，有三种状态：Pending（进行中）、Resolved（已完成，又称 Fulfilled）和 Rejected（已失败）。
一旦状态改变，就不会再变，任何时候都可以得到这个结果。Promise 对象的状态改变，只有两种可能：从 Pending 变为 Resolved 和从 Pending 变为 Rejected。只要这两种情况发生，状态就凝固了，不会再变了，会一直保持这个结果。就算改变已经发生了，你再对 Promise 对象添加回调函数，也会立即得到这个结果。这与事件（Event）完全不同，事件的特点是，如果你错过了它，再去监听，是得不到结果的。
有了 Promise 对象，就可以将异步操作以同步操作的流程表达出来，避免了层层嵌套的回调函数。此外，Promise 对象提供统一的接口，使得控制异步操作更加容易。
Promise 也有一些缺点。首先，无法取消 Promise，一旦新建它就会立即执行，无法中途取消。其次，如果不设置回调函数，Promise 内部抛出的错误，不会反应到外部。第三，当处于 Pending 状态时，无法得知目前进展到哪一个阶段（刚刚开始还是即将完成）。
ES6 规定，Promise 对象是一个构造函数，用来生成 Promise 实例。

下面代码创造了一个 Promise 实例。

```
var promise = new Promise(function(resolve, reject) {
  // ... some code

  if (/* 异步操作成功 */){
    resolve(value);
  } else {
    reject(error);
  }
});
```

Promise 构造函数接受一个函数作为参数，该函数的两个参数分别是 resolve 和 reject。它们是两个函数，由 JavaScript 引擎提供，不用自己部署。

resolve 函数的作用是，将 Promise 对象的状态从“未完成”变为“成功”（即从 Pending 变为 Resolved），在异步操作成功时调用，并将异步操作的结果，作为参数传递出去；reject 函数的作用是，将 Promise 对象的状态从“未完成”变为“失败”（即从 Pending 变为 Rejected），在异步操作失败时调用，并将异步操作报出的错误，作为参数传递出去。

Promise 实例生成以后，可以用 then 方法分别指定 Resolved 状态和 Reject 状态的回调函数。then 方法可以接受两个回调函数作为参数。第一个回调函数是 Promise 对象的状态变为 Resolved 时调用，第二个回调函数是 Promise 对象的状态变为 Reject 时调用。其中，第二个函数是可选的，不一定要提供。

用 Promise 对象封装 Ajax 操作

```
var getJSON = function(url) {
  var promise = new Promise(function(resolve, reject){
    var client = new XMLHttpRequest();
    client.open("GET", url);
    client.onreadystatechange = handler;
    client.responseType = "json";
    client.setRequestHeader("Accept", "application/json");
    client.send();

    function handler() {
      if (this.readyState !== 4) {
        return;
      }
      if (this.status === 200) {
        resolve(this.response);
      } else {
        reject(new Error(this.statusText));
      }
    };
  });

  return promise;
```

```
};

getJSON("/posts.json").then(function(json) {
  console.log('Contents: ' + json);
}, function(error) {
  console.error('出错了', error);
});
```

如果调用 resolve 函数和 reject 函数时带有参数，那么它们的参数会被传递给回调函数。reject 函数的参数通常是 Error 对象的实例，表示抛出的错误；resolve 函数的参数除了正常的值以外，还可能是另一个 Promise 实例，表示异步操作的结果有可能是一个值，也有可能是另一个异步操作，比如像下面这样。

```
var p1 = new Promise(function (resolve, reject) {
  // ...
});

var p2 = new Promise(function (resolve, reject) {
  // ...
  resolve(p1);
})
```