```
if (true) {
  let age = 26;
  console.log(age);   // 26
}
console.log(age);     // ReferenceError: age 没有定义
```

在这里，age 变量之所以不能在 if 块外部被引用，是因为它的作用域仅限于该块内部。块作用域是函数作用域的子集，因此适用于 var 的作用域限制同样也适用于 let。

let 也不允许同一个块作用域中出现冗余声明。这样会导致报错：

```
var name;
var name;

let age;
let age;  // SyntaxError；标识符 age 已经声明过了
```

当然，JavaScript 引擎会记录用于变量声明的标识符及其所在的块作用域，因此嵌套使用相同的标识符不会报错，而这是因为同一个块中没有重复声明：

```
var name = 'Nicholas';
console.log(name);    // 'Nicholas'
if (true) {
  var name = 'Matt';
  console.log(name);  // 'Matt'
}

let age = 30;
console.log(age);    // 30
if (true) {
  let age = 26;
  console.log(age);  // 26
}
```

对声明冗余报错不会因混用 let 和 var 而受影响。这两个关键字声明的并不是不同类型的变量，它们只是指出变量在相关作用域如何存在。

```
var name;
let name; // SyntaxError

let age;
var age; // SyntaxError
```

1. 暂时性死区

let 与 var 的另一个重要的区别，就是 let 声明的变量不会在作用域中被提升。

```
// name 会被提升
console.log(name); // undefined
var name = 'Matt';

// age 不会被提升
console.log(age); // ReferenceError: age 没有定义
let age = 26;
```

在解析代码时，JavaScript 引擎也会注意出现在块后面的 let 声明，只不过在此之前不能以任何方式来引用未声明的变量。在 let 声明之前的执行瞬间被称为“暂时性死区”（temporal dead zone），在此阶段引用任何后面才声明的变量都会抛出 ReferenceError。

2. 全局声明

与 var 关键字不同，使用 let 在全局作用域中声明的变量不会成为 window 对象的属性（var 声明的变量则会）。

```
var name = 'Matt';
console.log(window.name); // 'Matt'

let age = 26;
console.log(window.age);  // undefined
```

不过，let 声明仍然是在全局作用域中发生的，相应变量会在页面的生命周期内存续。因此，为了避免 SyntaxError，必须确保页面不会重复声明同一个变量。

3. 条件声明

在使用 var 声明变量时，由于声明会被提升，JavaScript 引擎会自动将多余的声明在作用域顶部合并为一个声明。因为 let 的作用域是块，所以不可能检查前面是否已经使用 let 声明过同名变量，同时也就不可能在没有声明的情况下声明它。

```
<script>
  var name = 'Nicholas';
  let age = 26;
</script>

<script>
  // 假设脚本不确定页面中是否已经声明了同名变量
  // 那它可以假设还没有声明过

  var name = 'Matt';
  // 这里没问题，因为可以被作为一个提升声明来处理
  // 不需要检查之前是否声明过同名变量

  let age = 36;
  // 如果 age 之前声明过，这里会报错
</script>
```

使用 try/catch 语句或 typeof 操作符也不能解决，因为条件块中 let 声明的作用域仅限于该块。

```
<script>
  let name = 'Nicholas';
  let age = 36;
</script>

<script>
  // 假设脚本不确定页面中是否已经声明了同名变量
  // 那它可以假设还没有声明过

  if (typeof name === 'undefined') {
    let name;
  }
  // name 被限制在 if {} 块的作用域内
  // 因此这个赋值形同全局赋值
  name = 'Matt';

  try {
    console.log(age); // 如果 age 没有声明过，则会报错
  }
  catch(error) {
    let age;
```

```
  }
  // age 被限制在 catch {}块的作用域内
  // 因此这个赋值形同全局赋值
  age = 26;
</script>
```

为此，对于 let 这个新的 ES6 声明关键字，不能依赖条件声明模式。

> **注意**　不能使用 let 进行条件式声明是件好事，因为条件声明是一种反模式，它让程序变得更难理解。如果你发现自己在使用这个模式，那一定有更好的替代方式。

4. for 循环中的 let 声明

在 let 出现之前，for 循环定义的迭代变量会渗透到循环体外部：

```
for (var i = 0; i < 5; ++i) {
  // 循环逻辑
}
console.log(i); // 5
```

改成使用 let 之后，这个问题就消失了，因为迭代变量的作用域仅限于 for 循环块内部：

```
for (let i = 0; i < 5; ++i) {
  // 循环逻辑
}
console.log(i); // ReferenceError: i 没有定义
```

在使用 var 的时候，最常见的问题就是对迭代变量的奇特声明和修改：

```
for (var i = 0; i < 5; ++i) {
    setTimeout(() => console.log(i), 0)
}
// 你可能以为会输出 0、1、2、3、4
// 实际上会输出 5、5、5、5、5
```

之所以会这样，是因为在退出循环时，迭代变量保存的是导致循环退出的值：5。在之后执行超时逻辑时，所有的 i 都是同一个变量，因而输出的都是同一个最终值。

而在使用 let 声明迭代变量时，JavaScript 引擎在后台会为每个迭代循环声明一个新的迭代变量。每个 setTimeout 引用的都是不同的变量实例，所以 console.log 输出的是我们期望的值，也就是循环执行过程中每个迭代变量的值。

```
for (let i = 0; i < 5; ++i) {
    setTimeout(() => console.log(i), 0)
}
// 会输出 0、1、2、3、4
```

这种每次迭代声明一个独立变量实例的行为适用于所有风格的 for 循环，包括 for-in 和 for-of 循环。

3.3.3　const 声明

const 的行为与 let 基本相同，唯一一个重要的区别是用它声明变量时必须同时初始化变量，且尝试修改 const 声明的变量会导致运行时错误。

```
const age = 26;
age = 36; // TypeError: 给常量赋值
```

```
// const 也不允许重复声明
const name = 'Matt';
const name = 'Nicholas'; // SyntaxError

// const 声明的作用域也是块
const name = 'Matt';
if (true) {
  const name = 'Nicholas';
}
console.log(name); // Matt
```


const 声明的限制只适用于它指向的变量的引用。换句话说，如果 const 变量引用的是一个对象，那么修改这个对象内部的属性并不违反 const 的限制。

```
const person = {};
person.name = 'Matt';  // ok
```

JavaScript 引擎会为 for 循环中的 let 声明分别创建独立的变量实例，虽然 const 变量跟 let 变量很相似，但是不能用 const 来声明迭代变量（因为迭代变量会自增）：

```
for (const i = 0; i < 10; ++i) {} // TypeError: 给常量赋值
```

不过，如果你只想用 const 声明一个不会被修改的 for 循环变量，那也是可以的。也就是说，每次迭代只是创建一个新变量。这对 for-of 和 for-in 循环特别有意义：

```
let i = 0;
for (const j = 7; i < 5; ++i) {
  console.log(j);
}
// 7, 7, 7, 7, 7

for (const key in {a: 1, b: 2}) {
  console.log(key);
}
// a, b

for (const value of [1,2,3,4,5]) {
  console.log(value);
}
// 1, 2, 3, 4, 5
```

3.3.4 声明风格及最佳实践

ECMAScript 6 增加 let 和 const 从客观上为这门语言更精确地声明作用域和语义提供了更好的支持。行为怪异的 var 所造成的各种问题，已经让 JavaScript 社区为之苦恼了很多年。随着这两个新关键字的出现，新的有助于提升代码质量的最佳实践也逐渐显现。

1. 不使用 var

有了 let 和 const，大多数开发者会发现自己不再需要 var 了。限制自己只使用 let 和 const 有助于提升代码质量，因为变量有了明确的作用域、声明位置，以及不变的值。

2. const 优先，let 次之

使用 const 声明可以让浏览器运行时强制保持变量不变，也可以让静态代码分析工具提前发现不合法的赋值操作。因此，很多开发者认为应该优先使用 const 来声明变量，只在提前知道未来会有修

改时，再使用 let。这样可以让开发者更有信心地推断某些变量的值永远不会变，同时也能迅速发现因意外赋值导致的非预期行为。

3.4　数据类型

ECMAScript 有 6 种简单数据类型（也称为**原始类型**）：Undefined、Null、Boolean、Number、String 和 Symbol。Symbol（符号）是 ECMAScript 6 新增的。还有一种复杂数据类型叫 Object（对象）。Object 是一种无序名值对的集合。因为在 ECMAScript 中不能定义自己的数据类型，所有值都可以用上述 7 种数据类型之一来表示。只有 7 种数据类型似乎不足以表示全部数据。但 ECMAScript 的数据类型很灵活，一种数据类型可以当作多种数据类型来使用。

3.4.1　typeof 操作符

因为 ECMAScript 的类型系统是松散的，所以需要一种手段来确定任意变量的数据类型。typeof 操作符就是为此而生的。对一个值使用 typeof 操作符会返回下列字符串之一：

- "undefined"表示值未定义；
- "boolean"表示值为布尔值；
- "string"表示值为字符串；
- "number"表示值为数值；
- "object"表示值为对象（而不是函数）或 null；
- "function"表示值为函数；
- "symbol"表示值为符号。

下面是使用 typeof 操作符的例子：

```
let message = "some string";
console.log(typeof message);    // "string"
console.log(typeof(message));   // "string"
console.log(typeof 95);         // "number"
```

在这个例子中，我们把一个变量（message）和一个数值字面量传给了 typeof 操作符。注意，因为 typeof 是一个操作符而不是函数，所以不需要参数（但可以使用参数）。

注意 typeof 在某些情况下返回的结果可能会让人费解，但技术上讲还是正确的。比如，调用 typeof null 返回的是"object"。这是因为特殊值 null 被认为是一个对空对象的引用。

> **注意**　严格来讲，函数在 ECMAScript 中被认为是对象，并不代表一种数据类型。可是，函数也有自己特殊的属性。为此，就有必要通过 typeof 操作符来区分函数和其他对象。

3.4.2　Undefined 类型

Undefined 类型只有一个值，就是特殊值 undefined。当使用 var 或 let 声明了变量但没有初始化时，就相当于给变量赋予了 undefined 值：

```
let message;
console.log(message == undefined); // true
```

在这个例子中，变量 message 在声明的时候并未初始化。而在比较它和 undefined 的字面值时，两者是相等的。这个例子等同于如下示例：

```
let message = undefined;
console.log(message == undefined); // true
```

这里，变量 message 显式地以 undefined 来初始化。但这是不必要的，因为默认情况下，任何未经初始化的变量都会取得 undefined 值。

> **注意** 一般来说，永远不用显式地给某个变量设置 undefined 值。字面值 undefined 主要用于比较，而且在 ECMA-262 第 3 版之前是不存在的。增加这个特殊值的目的就是为了正式明确空对象指针（null）和未初始化变量的区别。

注意，包含 undefined 值的变量跟未定义变量是有区别的。请看下面的例子：

```
let message;    // 这个变量被声明了，只是值为 undefined

// 确保没有声明过这个变量
// let age

console.log(message); // "undefined"
console.log(age);     // 报错
```

在上面的例子中，第一个 console.log 会指出变量 message 的值，即"undefined"。而第二个 console.log 要输出一个未声明的变量 age 的值，因此会导致报错。对未声明的变量，只能执行一个有用的操作，就是对它调用 typeof。（对未声明的变量调用 delete 也不会报错，但这个操作没什么用，实际上在严格模式下会抛出错误。）

在对未初始化的变量调用 typeof 时，返回的结果是"undefined"，但对未声明的变量调用它时，返回的结果还是"undefined"，这就有点让人看不懂了。比如下面的例子：

```
let message; // 这个变量被声明了，只是值为 undefined

// 确保没有声明过这个变量
// let age

console.log(typeof message); // "undefined"
console.log(typeof age);     // "undefined"
```

无论是声明还是未声明，typeof 返回的都是字符串"undefined"。逻辑上讲这是对的，因为虽然严格来讲这两个变量存在根本性差异，但它们都无法执行实际操作。

> **注意** 即使未初始化的变量会被自动赋予 undefined 值，但我们仍然建议在声明变量的同时进行初始化。这样，当 typeof 返回"undefined"时，你就会知道那是因为给定的变量尚未声明，而不是声明了但未初始化。

undefined 是一个假值。因此，如果需要，可以用更简洁的方式检测它。不过要记住，也有很多其他可能的值同样是假值。所以一定要明确自己想检测的就是 undefined 这个字面值，而不仅仅是假值。

```
let message; // 这个变量被声明了，只是值为 undefined
// age 没有声明

if (message) {
  // 这个块不会执行
}

if (!message) {
  // 这个块会执行
}

if (age) {
  // 这里会报错
}
```

3.4.3 Null 类型

Null 类型同样只有一个值，即特殊值 null。逻辑上讲，null 值表示一个空对象指针，这也是给 typeof 传一个 null 会返回"object"的原因：

```
let car = null;
console.log(typeof car);  // "object"
```

在定义将来要保存对象值的变量时，建议使用 null 来初始化，不要使用其他值。这样，只要检查这个变量的值是不是 null 就可以知道这个变量是否在后来被重新赋予了一个对象的引用，比如：

```
if (car != null) {
  // car 是一个对象的引用
}
```

undefined 值是由 null 值派生而来的，因此 ECMA-262 将它们定义为表面上相等，如下面的例子所示：

```
console.log(null == undefined);  // true
```

用等于操作符（==）比较 null 和 undefined 始终返回 true。但要注意，这个操作符会为了比较而转换它的操作数（本章后面将详细介绍）。

即使 null 和 undefined 有关系，它们的用途也是完全不一样的。如前所述，永远不必显式地将变量值设置为 undefined。但 null 不是这样的。任何时候，只要变量要保存对象，而当时又没有那个对象可保存，就要用 null 来填充该变量。这样就可以保持 null 是空对象指针的语义，并进一步将其与 undefined 区分开来。

null 是一个假值。因此，如果需要，可以用更简洁的方式检测它。不过要记住，也有很多其他可能的值同样是假值。所以一定要明确自己想检测的就是 null 这个字面值，而不仅仅是假值。

```
let message = null;
let age;

if (message) {
  // 这个块不会执行
}

if (!message) {
  // 这个块会执行
}
```

```
if (age) {
  // 这个块不会执行
}

if (!age) {
  // 这个块会执行
}
```


3.4.4 Boolean 类型

Boolean（布尔值）类型是 ECMAScript 中使用最频繁的类型之一，有两个字面值：true 和 false。这两个布尔值不同于数值，因此 true 不等于 1，false 不等于 0。下面是给变量赋布尔值的例子：

```
let found = true;
let lost = false;
```

注意，布尔值字面量 true 和 false 是区分大小写的，因此 True 和 False（及其他大小混写形式）是有效的标识符，但不是布尔值。

虽然布尔值只有两个，但所有其他 ECMAScript 类型的值都有相应布尔值的等价形式。要将一个其他类型的值转换为布尔值，可以调用特定的 Boolean()转型函数：

```
let message = "Hello world!";
let messageAsBoolean = Boolean(message);
```

在这个例子中，字符串 message 会被转换为布尔值并保存在变量 messageAsBoolean 中。Boolean()转型函数可以在任意类型的数据上调用，而且始终返回一个布尔值。什么值能转换为 true 或 false 的规则取决于数据类型和实际的值。下表总结了不同类型与布尔值之间的转换规则。

数据类型	转换为 true 的值	转换为 false 的值
Boolean	true	false
String	非空字符串	""（空字符串）
Number	非零数值（包括无穷值）	0、NaN（参见后面的相关内容）
Object	任意对象	null
Undefined	N/A（不存在）	undefined

理解以上转换非常重要，因为像 if 等流控制语句会自动执行其他类型值到布尔值的转换，例如：

```
let message = "Hello world!";
if (message) {
  console.log("Value is true");
}
```

在这个例子中，console.log 会输出字符串"Value is true"，因为字符串 message 会被自动转换为等价的布尔值 true。由于存在这种自动转换，理解流控制语句中使用的是什么变量就非常重要。错误地使用对象而不是布尔值会明显改变应用程序的执行流。

3.4.5 Number 类型

ECMAScript 中最有意思的数据类型或许就是 Number 了。Number 类型使用 IEEE 754 格式表示整数和浮点值（在某些语言中也叫双精度值）。不同的数值类型相应地也有不同的数值字面量格式。

最基本的数值字面量格式是十进制整数，直接写出来即可：

```
let intNum = 55;  // 整数
```

整数也可以用八进制（以 8 为基数）或十六进制（以 16 为基数）字面量表示。对于八进制字面量，第一个数字必须是零（0），然后是相应的八进制数字（数值 0~7）。如果字面量中包含的数字超出了应有的范围，就会忽略前缀的零，后面的数字序列会被当成十进制数，如下所示：

```
let octalNum1 = 070;  // 八进制的 56
let octalNum2 = 079;  // 无效的八进制值，当成 79 处理
let octalNum3 = 08;   // 无效的八进制值，当成 8 处理
```

八进制字面量在严格模式下是无效的，会导致 JavaScript 引擎抛出语法错误。[①]

要创建十六进制字面量，必须让真正的数值前缀 0x（区分大小写），然后是十六进制数字（0~9 以及 A~F）。十六进制数字中的字母大小写均可。下面是几个例子：

```
let hexNum1 = 0xA;   // 十六进制 10
let hexNum2 = 0x1f;  // 十六进制 31
```

使用八进制和十六进制格式创建的数值在所有数学操作中都被视为十进制数值。

> **注意**　由于 JavaScript 保存数值的方式，实际中可能存在正零（+0）和负零（–0）。正零和负零在所有情况下都被认为是等同的，这里特地说明一下。

1. 浮点值

要定义浮点值，数值中必须包含小数点，而且小数点后面必须至少有一个数字。虽然小数点前面不是必须有整数，但推荐加上。下面是几个例子：

```
let floatNum1 = 1.1;
let floatNum2 = 0.1;
let floatNum3 = .1;   // 有效，但不推荐
```

因为存储浮点值使用的内存空间是存储整数值的两倍，所以 ECMAScript 总是想方设法把值转换为整数。在小数点后面没有数字的情况下，数值就会变成整数。类似地，如果数值本身就是整数，只是小数点后面跟着 0（如 1.0），那它也会被转换为整数，如下例所示：

```
let floatNum1 = 1.;   // 小数点后面没有数字，当成整数 1 处理
let floatNum2 = 10.0; // 小数点后面是零，当成整数 10 处理
```

对于非常大或非常小的数值，浮点值可以用科学记数法来表示。科学记数法用于表示一个应该乘以 10 的给定次幂的数值。ECMAScript 中科学记数法的格式要求是一个数值（整数或浮点数）后跟一个大写或小写的字母 e，再加上一个要乘的 10 的多少次幂。比如：

```
let floatNum = 3.125e7; // 等于 31250000
```

在这个例子中，floatNum 等于 31 250 000，只不过科学记数法显得更简洁。这种表示法实际上相当于说：“以 3.125 作为系数，乘以 10 的 7 次幂。”

科学记数法也可以用于表示非常小的数值，例如 0.000 000 000 000 000 03。这个数值用科学记数法可以表示为 3e–17。默认情况下，ECMAScript 会将小数点后至少包含 6 个零的浮点值转换为科学记数法

---

① ECMAScript 2015 或 ES6 中的八进制值通过前缀 0o 来表示；严格模式下，前缀 0 会被视为语法错误，如果要表示八进制值，应该使用前缀 0o。——译者注

（例如，0.000 000 3 会被转换为 3e–7）。

浮点值的精确度最高可达 17 位小数，但在算术计算中远不如整数精确。例如，0.1 加 0.2 得到的不是 0.3，而是 0.300 000 000 000 000 04。由于这种微小的舍入错误，导致很难测试特定的浮点值。比如下面的例子：

```
if (a + b == 0.3) {       // 别这么干!
  console.log("You got 0.3.");
}
```

这里检测两个数值之和是否等于 0.3。如果两个数值分别是 0.05 和 0.25，或者 0.15 和 0.15，那没问题。但如果是 0.1 和 0.2，如前所述，测试将失败。因此永远不要测试某个特定的浮点值。

> **注意** 之所以存在这种舍入错误，是因为使用了 IEEE 754 数值，这种错误并非 ECMAScript 所独有。其他使用相同格式的语言也有这个问题。

2. 值的范围

由于内存的限制，ECMAScript 并不支持表示这个世界上的所有数值。ECMAScript 可以表示的最小数值保存在 `Number.MIN_VALUE` 中，这个值在多数浏览器中是 5e–324；可以表示的最大数值保存在 `Number.MAX_VALUE` 中，这个值在多数浏览器中是 1.797 693 134 862 315 7e+308。如果某个计算得到的数值结果超出了 JavaScript 可以表示的范围，那么这个数值会被自动转换为一个特殊的 `Infinity`（无穷）值。任何无法表示的负数以`-Infinity`（负无穷大）表示，任何无法表示的正数以 `Infinity`（正无穷大）表示。

如果计算返回正 `Infinity` 或负 `Infinity`，则该值将不能再进一步用于任何计算。这是因为 `Infinity` 没有可用于计算的数值表示形式。要确定一个值是不是有限大（即介于 JavaScript 能表示的最小值和最大值之间），可以使用 `isFinite()`函数，如下所示：

```
let result = Number.MAX_VALUE + Number.MAX_VALUE;
console.log(isFinite(result));  // false
```

虽然超出有限数值范围的计算并不多见，但总归还是有可能的。因此在计算非常大或非常小的数值时，有必要监测一下计算结果是否超出范围。

> **注意** 使用 `Number.NEGATIVE_INFINITY` 和 `Number.POSITIVE_INFINITY` 也可以获取正、负 `Infinity`。没错，这两个属性包含的值分别就是`-Infinity` 和 `Infinity`。

3. NaN

有一个特殊的数值叫 `NaN`，意思是“不是数值”（Not a Number），用于表示本来要返回数值的操作失败了（而不是抛出错误）。比如，用 0 除任意数值在其他语言中通常都会导致错误，从而中止代码执行。但在 ECMAScript 中，0、+0 或–0 相除会返回 `NaN`：

```
console.log(0/0);     // NaN
console.log(-0/+0);   // NaN
```

如果分子是非 0 值，分母是有符号 0 或无符号 0，则会返回 `Infinity` 或`-Infinity`：

```
console.log(5/0);    // Infinity
console.log(5/-0);   // -Infinity
```