变量声明

let和const是JavaScript里相对较新的变量声明方式。
像我们之前提到过的，let在很多方面与var是相似的，但是可以帮助大家避免在JavaScript里常见一些问题。
const是对let的一个增强，它能阻止对一个变量再次赋值。

因为TypeScript是JavaScript的超集，所以它本身就支持let和const。
下面我们会详细说明这些新的声明方式以及为什么推荐使用它们来代替var。

如果你之前使用JavaScript时没有特别在意，那么这节内容会唤起你的回忆。
如果你已经对var声明的怪异之处了如指掌，那么你可以轻松地略过这节。

var 声明

一直以来我们都是通过var关键字定义JavaScript变量。

```
var a = 10;
```

大家都能理解，这里定义了一个名为a值为10的变量。

我们也可以在函数内部定义变量：

```
function f() {
    var message = "Hello, world!";

    return message;
}
```

并且我们也可以在其它函数内部访问相同的变量。

```
function f() {
    var a = 10;
    return function g() {
        var b = a + 1;
        return b;
    }
}

var g = f();
g(); // returns 11;
```

上面的例子里，g可以获取到f函数里定义的a变量。
每当g被调用时，它都可以访问到f里的a变量。
即使当g在f已经执行完后才被调用，它仍然可以访问及修改a。

```
function f() {
    var a = 1;

    a = 2;
    var b = g();
```

```
    a = 3;

    return b;

    function g() {
        return a;
    }
}

f(); // returns 2
```

作用域规则

对于熟悉其它语言的人来说，`var`声明有些奇怪的作用域规则。
看下面的例子：

```
function f(shouldInitialize: boolean) {
    if (shouldInitialize) {
        var x = 10;
    }

    return x;
}

f(true);  // returns '10'
f(false); // returns 'undefined'
```

有些读者可能要多看几遍这个例子。
变量`x`是定义在*`if`语句里面*，但是我们却可以在语句的外面访问它。
这是因为`var`声明可以在包含它的函数，模块，命名空间或全局作用域内部任何位置被访问（我们后面会详细介绍），包含它的代码块对此没有什么影响。
有些人称此为*`var`作用域*或*函数作用域*。
函数参数也使用函数作用域。

这些作用域规则可能会引发一些错误。
其中之一就是，多次声明同一个变量并不会报错：

```
function sumMatrix(matrix: number[][]) {
    var sum = 0;
    for (var i = 0; i < matrix.length; i++) {
        var currentRow = matrix[i];
        for (var i = 0; i < currentRow.length; i++) {
            sum += currentRow[i];
        }
    }

    return sum;
}
```

这里很容易看出一些问题，里层的`for`循环会覆盖变量`i`，因为所有`i`都引用相同的函数作用域内的变量。
有经验的开发者们很清楚，这些问题可能在代码审查时漏掉，引发无穷的麻烦。

变量获取怪异之处

快速的猜一下下面的代码会返回什么：

```
for (var i = 0; i < 10; i++) {
    setTimeout(function() { console.log(i); }, 100 * i);
}
```

介绍一下，setTimeout会在若干毫秒的延时后执行一个函数（等待其它代码执行完毕）。

好吧，看一下结果：

```
10
10
10
10
10
10
10
10
10
10
```

很多JavaScript程序员对这种行为已经很熟悉了，但如果你很不解，你并不是一个人。
大多数人期望输出结果是这样：

```
0
1
2
3
4
5
6
7
8
9
```

还记得我们上面讲的变量获取吗？

每当g被调用时，它都可以访问到f里的a变量。

让我们花点时间考虑在这个上下文里的情况。
setTimeout在若干毫秒后执行一个函数，并且是在for循环结束后。
for循环结束后，i的值为10。
所以当函数被调用的时候，它会打印出10！

一个通常的解决方法是使用立即执行的函数表达式（IIFE）来捕获每次迭代时i的值：

```
for (var i = 0; i < 10; i++) {
    // capture the current state of 'i'
    // by invoking a function with its current value
    (function(i) {
        setTimeout(function() { console.log(i); }, 100 * i);
    })(i);
}
```

这种奇怪的形式我们已经司空见惯了。
参数i会覆盖for循环里的i，但是因为我们起了同样的名字，所以我们不用怎么改for循环

体里的代码。

`let` 声明

现在你已经知道了`var`存在一些问题，这恰好说明了为什么用`let`语句来声明变量。
除了名字不同外，`let`与`var`的写法一致。

```
let hello = "Hello!";
```

主要的区别不在语法上，而是语义，我们接下来会深入研究。

块作用域

当用`let`声明一个变量，它使用的是*词法作用域*或*块作用域*。
不同于使用`var`声明的变量那样可以在包含它们的函数外访问，块作用域变量在包含它们的块或`for`循环之外是不能访问的。

```
function f(input: boolean) {
    let a = 100;

    if (input) {
        // Still okay to reference 'a'
        let b = a + 1;
        return b;
    }

    // Error: 'b' doesn't exist here
    return b;
}
```

这里我们定义了2个变量`a`和`b`。
`a`的作用域是`f`函数体内，而`b`的作用域是`if`语句块里。

在`catch`语句里声明的变量也具有同样的作用域规则。

```
try {
    throw "oh no!";
}
catch (e) {
    console.log("Oh well.");
}

// Error: 'e' doesn't exist here
console.log(e);
```

拥有块级作用域的变量的另一个特点是，它们不能在被声明之前读或写。
虽然这些变量始终“存在”于它们的作用域里，但在直到声明它的代码之前的区域都属于*暂时性死区*。
它只是用来说明我们不能在`let`语句之前访问它们，幸运的是TypeScript可以告诉我们这些信息。

```
a++; // illegal to use 'a' before it's declared;
let a;
```

注意一点，我们仍然可以在一个拥有块作用域变量被声明前*获取*它。
只是我们不能在变量声明前去调用那个函数。
如果生成代码目标为ES2015，现代的运行时会抛出一个错误；然而，现今TypeScript是不会报错的。

```
function foo() {
    // okay to capture 'a'
    return a;
}

// 不能在'a'被声明前调用'foo'
// 运行时应该抛出错误
foo();

let a;
```

关于*暂时性死区*的更多信息，查看这里Mozilla Developer Network.

重定义及屏蔽

我们提过使用`var`声明时，它不在乎你声明多少次；你只会得到1个。

```
function f(x) {
    var x;
    var x;

    if (true) {
        var x;
    }
}
```

在上面的例子里，所有`x`的声明实际上都引用一个*相同*的`x`，并且这是完全有效的代码。
这经常会成为bug的来源。
好的是，`let`声明就不会这么宽松了。

```
let x = 10;
let x = 20; // 错误，不能在1个作用域里多次声明`x`
```

并不是要求两个均是块级作用域的声明TypeScript才会给出一个错误的警告。

```
function f(x) {
    let x = 100; // error: interferes with parameter declaration
}

function g() {
    let x = 100;
    var x = 100; // error: can't have both declarations of 'x'
}
```

并不是说块级作用域变量不能在函数作用域内声明。
而是块级作用域变量需要在不用的块里声明。

```
function f(condition, x) {
    if (condition) {
        let x = 100;
        return x;
```

```
    }

    return x;
}

f(false, 0); // returns 0
f(true, 0);  // returns 100
```

在一个嵌套作用域里引入一个新名字的行为称做*屏蔽*。
它是一把双刃剑，它可能会不小心地引入新问题，同时也可能会解决一些错误。
例如，假设我们现在用`let`重写之前的`sumMatrix`函数。

```
function sumMatrix(matrix: number[][]) {
    let sum = 0;
    for (let i = 0; i < matrix.length; i++) {
        var currentRow = matrix[i];
        for (let i = 0; i < currentRow.length; i++) {
            sum += currentRow[i];
        }
    }

    return sum;
}
```

这个版本的循环能得到正确的结果，因为内层循环的`i`可以屏蔽掉外层循环的`i`。

*通常*来讲应该避免使用屏蔽，因为我们需要写出清晰的代码。
同时也有些场景适合利用它，你需要好好打算一下。

块级作用域变量的获取

在我们最初谈及获取用`var`声明的变量时，我们简略地探究了一下在获取到了变量之后它的行为是怎样的。
直观地讲，每次进入一个作用域时，它创建了一个变量的*环境*。
就算作用域内代码已经执行完毕，这个环境与其捕获的变量依然存在。

```
function theCityThatAlwaysSleeps() {
    let getCity;

    if (true) {
        let city = "Seattle";
        getCity = function() {
            return city;
        }
    }

    return getCity();
}
```

因为我们已经在`city`的环境里获取到了`city`，所以就算`if`语句执行结束后我们仍然可以访问它。

回想一下前面`setTimeout`的例子，我们最后需要使用立即执行的函数表达式来获取每次`for`循环迭代里的状态。
实际上，我们做的是为获取到的变量创建了一个新的变量环境。

这样做挺痛苦的，但是幸运的是，你不必在TypeScript里这样做了。

当`let`声明出现在循环体里时拥有完全不同的行为。
不仅是在循环里引入了一个新的变量环境，而是针对*每次迭代*都会创建这样一个新作用域。
这就是我们在使用立即执行的函数表达式时做的事，所以在`setTimeout`例子里我们仅使用`let`声明就可以了。

```
for (let i = 0; i < 10 ; i++) {
    setTimeout(function() {console.log(i); }, 100 * i);
}
```

会输出与预料一致的结果：

```
0
1
2
3
4
5
6
7
8
9
```

const 声明

`const` 声明是声明变量的另一种方式。

```
const numLivesForCat = 9;
```

它们与`let`声明相似，但是就像它的名字所表达的，它们被赋值后不能再改变。
换句话说，它们拥有与`let`相同的作用域规则，但是不能对它们重新赋值。

这很好理解，它们引用的值是*不可变的*。

```
const numLivesForCat = 9;
const kitty = {
    name: "Aurora",
    numLives: numLivesForCat,
}

// Error
kitty = {
    name: "Danielle",
    numLives: numLivesForCat
};

// all "okay"
kitty.name = "Rory";
kitty.name = "Kitty";
kitty.name = "Cat";
kitty.numLives--;
```

除非你使用特殊的方法去避免，实际上`const`变量的内部状态是可修改的。

幸运的是，TypeScript允许你将对象的成员设置成只读的。
接口一章有详细说明。

`let` vs. `const`

现在我们有两种作用域相似的声明方式，我们自然会问到底应该使用哪个。
与大多数泛泛的问题一样，答案是：依情况而定。

使用最小特权原则，所有变量除了你计划去修改的都应该使用`const`。
基本原则就是如果一个变量不需要对它写入，那么其它使用这些代码的人也不能够写入它们，并且要思考为什么会需要对这些变量重新赋值。
使用`const`也可以让我们更容易的推测数据的流动。

另一方面，用户很喜欢`let`的简洁性。
这个手册大部分地方都使用了`let`。

跟据你的自己判断，如果合适的话，与团队成员商议一下。

解构

Another TypeScript已经可以解析其它 ECMAScript 2015 特性了。
完整列表请参见 the article on the Mozilla Developer Network。
本章，我们将给出一个简短的概述。

解构数组

最简单的解构莫过于数组的解构赋值了：

```
let input = [1, 2];
let [first, second] = input;
console.log(first); // outputs 1
console.log(second); // outputs 2
```

这创建了2个命名变量 `first` 和 `second`。
相当于使用了索引，但更为方便：

```
first = input[0];
second = input[1];
```

解构作用于已声明的变量会更好：

```
// swap variables
[first, second] = [second, first];
```

作用于函数参数：

```
function f([first, second]: [number, number]) {
    console.log(first);
    console.log(second);
}
```

```
f(input);
```

你可以在数组里使用...语法创建剩余变量：

```
let [first, ...rest] = [1, 2, 3, 4];
console.log(first); // outputs 1
console.log(rest); // outputs [ 2, 3, 4 ]
```

当然，由于是JavaScript, 你可以忽略你不关心的尾随元素：

```
let [first] = [1, 2, 3, 4];
console.log(first); // outputs 1
```

或其它元素：

```
let [, second, , fourth] = [1, 2, 3, 4];
```

对象解构

你也可以解构对象：

```
let o = {
    a: "foo",
    b: 12,
    c: "bar"
};
let { a, b } = o;
```

这通过 `o.a` and `o.b` 创建了 `a` 和 `b` 。
注意，如果你不需要 `c` 你可以忽略它。

就像数组解构，你可以用没有声明的赋值：

```
({ a, b } = { a: "baz", b: 101 });
```

注意，我们需要用括号将它括起来，因为Javascript通常会将以 `{` 起始的语句解析为一个块。

你可以在对象里使用...语法创建剩余变量：

```
let { a, ...passthrough } = o;
let total = passthrough.b + passthrough.c.length;
```

属性重命名

你也可以给属性以不同的名字：

```
let { a: newName1, b: newName2 } = o;
```

这里的语法开始变得混乱。
你可以将 `a: newName1` 读做 "`a` 作为 `newName1`"。
方向是从左到右，好像你写成了以下样子：

```
let newName1 = o.a;
```

```
let newName2 = o.b;
```

令人困惑的是，这里的冒号*不是*指示类型的。
如果你想指定它的类型， 仍然需要在其后写上完整的模式。

```
let {a, b}: {a: string, b: number} = o;
```

默认值

默认值可以让你在属性为 undefined 时使用缺省值：

```
function keepWholeObject(wholeObject: { a: string, b?: number }) {
    let { a, b = 1001 } = wholeObject;
}
```

现在，即使 `b` 为 undefined， `keepWholeObject` 函数的变量 `wholeObject` 的属性 `a` 和 `b` 都会有值。

函数声明

解构也能用于函数声明。
看以下简单的情况：

```
type C = { a: string, b?: number }
function f({ a, b }: C): void {
    // ...
}
```

但是，通常情况下更多的是指定默认值，解构默认值有些棘手。
首先，你需要知道在设置默认值之前设置其类型。

```
function f({ a, b } = { a: "", b: 0 }): void {
    // ...
}
f(); // ok, default to { a: "", b: 0 }
```

其次，你需要知道在解构属性上给予一个默认或可选的属性用来替换主初始化列表。
要知道 `C` 的定义有一个 `b` 可选属性：

```
function f({ a, b = 0 } = { a: "" }): void {
    // ...
}
f({ a: "yes" }); // ok, default b = 0
f(); // ok, default to {a: ""}, which then defaults b = 0
f({}); // error, 'a' is required if you supply an argument
```

要小心使用解构。
从前面的例子可以看出，就算是最简单的解构表达式也是难以理解的。
尤其当存在深层嵌套解构的时候，就算这时没有堆叠在一起的重命名，默认值和类型注解，也是令人难以理解的。
解构表达式要尽量保持小而简单。
你自己也可以直接使用解构将会生成的赋值表达式。

展开

展开操作符正与解构相反。
它允许你将一个数组展开为另一个数组，或将一个对象展开为另一个对象。
例如：

```
let first = [1, 2];
let second = [3, 4];
let bothPlus = [0, ...first, ...second, 5];
```

这会令`bothPlus`的值为`[0, 1, 2, 3, 4, 5]`。
展开操作创建了`first`和`second`的一份浅拷贝。
它们不会被展开操作所改变。

你还可以展开对象：

```
let defaults = { food: "spicy", price: "$", ambiance: "noisy" };
let search = { ...defaults, food: "rich" };
```

`search`的值为`{ food: "rich", price: "$", ambiance: "noisy" }`。
对象的展开比数组的展开要复杂的多。
像数组展开一样，它是从左至右进行处理，但结果仍为对象。
这就意味着出现在展开对象后面的属性会覆盖前面的属性。
因此，如果我们修改上面的例子，在结尾处进行展开的话：

```
let defaults = { food: "spicy", price: "$", ambiance: "noisy" };
let search = { food: "rich", ...defaults };
```

那么，`defaults`里的`food`属性会重写`food: "rich"`，在这里这并不是我们想要的结果。

对象展开还有其它一些意想不到的限制。
首先，它只包含自身的可枚举的属性。
首先，当你展开一个对象实例时，你会丢失其方法：

```
class C {
  p = 12;
  m() {
  }
}
let c = new C();
let clone = { ...c };
clone.p; // ok
clone.m(); // error!
```

其次，TypeScript编译器不允许展开泛型函数上的类型参数。
这个特性会在TypeScript的未来版本中考虑实现。