提前使用变量

ES6 规范定义了一个新概念，叫作 TDZ（Temporal Dead Zone，暂时性死区）。

TDZ 指的是由于代码中的变量还没有初始化而不能被引用的情况。

对此，最直观的例子是 ES6 规范中的 `let` 块作用域：

```
{
    a = 2;      // ReferenceError!
    let a;
}
```

`a = 2` 试图在 `let a` 初始化 `a` 之前使用该变量（其作用域在 `{ .. }` 内），这里就是 `a` 的 TDZ，会产生错误。

有意思的是，对未声明变量使用 `typeof` 不会产生错误（参见第 1 章），但在 TDZ 中却会报错：

```
{
    typeof a;   // undefined
    typeof b;   // ReferenceError! (TDZ)
    let b;
}
```

5.5 函数参数

另一个 TDZ 违规的例子是 ES6 中的参数默认值（参见本系列的《你不知道的 JavaScript（下卷）》的“ES6 & Beyond”部分）：

```
var b = 3;

function foo( a = 42, b = a + b + 5 ) {
    // ..
}
```

`b = a + b + 5` 在参数 `b`（= 右边的 `b`，而不是函数外的那个）的 TDZ 中访问 `b`，所以会出错。而访问 `a` 却没有问题，因为此时刚好跨出了参数 `a` 的 TDZ。

在 ES6 中，如果参数被省略或者值为 `undefined`，则取该参数的默认值：

```
function foo( a = 42, b = a + 1 ) {
    console.log( a, b );
}

foo();                  // 42 43
foo( undefined );       // 42 43
foo( 5 );               // 5 6
foo( void 0, 7 );       // 42 7
foo( null );            // null 1
```

表达式 `a + 1` 中 `null` 被强制类型转换为 `0`。详情请参见第 4 章。

对 ES6 中的参数默认值而言，参数被省略或被赋值为 `undefined` 效果都一样，都是取该参数的默认值。然而某些情况下，它们之间还是有区别的：

```
function foo( a = 42, b = a + 1 ) {
    console.log(
        arguments.length, a, b,
        arguments[0], arguments[1]
    );
}

foo();                  // 0 42 43 undefined undefined
foo( 10 );              // 1 10 11 10 undefined
foo( 10, undefined );   // 2 10 11 10 undefined
foo( 10, null );        // 2 10 null 10 null
```

虽然参数 `a` 和 `b` 都有默认值，但是函数不带参数时，`arguments` 数组为空。

相反，如果向函数传递 `undefined` 值，则 `arguments` 数组中会出现一个值为 `undefined` 的单元，而不是默认值。

ES6 参数默认值会导致 `arguments` 数组和相对应的命名参数之间出现偏差，ES5 也会出现这种情况：

```
function foo(a) {
    a = 42;
    console.log( arguments[0] );
}

foo( 2 );   // 42 (linked)
foo();      // undefined (not linked)
```

向函数传递参数时，`arguments` 数组中的对应单元会和命名参数建立关联（linkage）以得到相同的值。相反，不传递参数就不会建立关联。

但是，在严格模式中并没有建立关联这一说：

```
function foo(a) {
    "use strict";
    a = 42;
    console.log( arguments[0] );
}

foo( 2 );   // 2 (not linked)
foo();      // undefined (not linked)
```

因此，在开发中不要依赖这种关联机制。实际上，它是 JavaScript 语言引擎底层实现的一个抽象泄漏（leaky abstraction），并不是语言本身的特性。

arguments 数组已经被废止（特别是在 ES6 引入剩余参数 ... 之后，参见本系列的《你不知道的 JavaScript（下卷）》的“ES6 & Beyond”部分），不过它并非一无是处。

在 ES6 之前，获得函数所有参数的唯一途径就是 arguments 数组。此外，即使将命名参数和 arguments 数组混用也不会出错，只需遵守一个原则，即*不要同时访问命名参数和其对应的 arguments 数组单元*。

```
function foo(a) {
    console.log( a + arguments[1] ); // 安全!
}

foo( 10, 32 );  // 42
```

5.6 try..finally

try..catch 对我们来说可能已经非常熟悉了。但你是否知道 try 可以和 catch 或者 finally 配对使用，并且必要时两者可同时出现？

finally 中的代码总是会在 try 之后执行，如果有 catch 的话则在 catch 之后执行。也可以将 finally 中的代码看作一个回调函数，即无论出现什么情况最后一定会被调用。

如果 try 中有 return 语句会出现什么情况呢？ return 会返回一个值，那么调用该函数并得到返回值的代码是在 finally 之前还是之后执行呢？

```
function foo() {
        try {
                return 42;
        }
        finally {
                console.log( "Hello" );
        }

        console.log( "never runs" );
}

console.log( foo() );
// Hello
// 42
```

这里 return 42 先执行，并将 foo() 函数的返回值设置为 42。然后 try 执行完毕，接着执行 finally。最后 foo() 函数执行完毕，console.log(..) 显示返回值。

try 中的 throw 也是如此：

```
 function foo() {
        try {
                throw 42;
        }
        finally {
                console.log( "Hello" );
        }

        console.log( "never runs" );
}

console.log( foo() );
// Hello
// Uncaught Exception: 42
```

如果 finally 中抛出异常（无论是有意还是无意），函数就会在此处终止。如果此前 try 中已经有 return 设置了返回值，则该值会被丢弃：

```
function foo() {
        try {
                return 42;
        }
        finally {
                throw "Oops!";
        }

        console.log( "never runs" );
}

console.log( foo() );
// Uncaught Exception: Oops!
```

continue 和 break 等控制语句也是如此：

```
for (var i=0; i<10; i++) {
        try {
                continue;
        }
        finally {
                console.log( i );
        }
}
// 0 1 2 3 4 5 6 7 8 9
```

continue 在每次循环之后，会在 i++ 执行之前执行 console.log(i)，所以结果是 0..9 而非 1..10。

ES6 中新加入了 yield（参见本书的“异步和性能”部分），可以将其视为 return 的中间版本。然而与 return 不同的是，yield 在 generator（ES6 的另一个新特性）重新开始时才结束，这意味着 try { .. yield .. } 并未结束，因此 finally 不会在 yield 之后立即执行。

finally 中的 return 会覆盖 try 和 catch 中 return 的返回值：

```
function foo() {
        try {
                return 42;
        }
        finally {
                // 没有返回语句,所以没有覆盖
        }
}

function bar() {
        try {
                return 42;
        }
        finally {
                // 覆盖前面的 return 42
                return;
        }
}

function baz() {
        try {
                return 42;
        }
        finally {
                // 覆盖前面的 return 42
                return "Hello";
        }
}

foo();  // 42
bar();  // undefined
baz();  // Hello
```

通常来说，在函数中省略 return 的结果和 return; 及 return undefined; 是一样的，但是在 finally 中省略 return 则会返回前面的 return 设定的返回值。

事实上，还可以将 finally 和带标签的 break 混合使用（参见 5.1.3 节）。例如：

```
function foo() {
        bar: {
                try {
                        return 42;
                }
                finally {
                        // 跳出标签为bar的代码块
                        break bar;
                }
        }

        console.log( "Crazy" );
```

```
        return "Hello";
}

console.log( foo() );
// Crazy
// Hello
```

但切勿这样操作。利用 finally 加带标签的 break 来跳过 return 只会让代码变得晦涩难懂，即使加上注释也是如此。

5.7 switch

现在来简单介绍一下 switch，可以把它看作 if..else if..else.. 的简化版本：

```
switch (a) {
        case 2:
                // 执行一些代码
                break;
        case 42:
                // 执行另外一些代码
                break;
        default:
                // 执行缺省代码
}
```

这里 a 与 case 表达式逐一进行比较。如果匹配就执行该 case 中的代码，直到 break 或者 switch 代码块结束。

这看似并无特别之处，但其中存在一些不太为人所知的陷阱。

首先，a 和 case 表达式的匹配算法与 ===（参见第 4 章）相同。通常 case 语句中的 switch 都是简单值，所以这并没有问题。

然而，有时可能会需要通过强制类型转换来进行相等比较（即 ==，参见第 4 章），这时就需要做一些特殊处理：

```
var a = "42";

switch (true) {
        case a == 10:
                console.log( "10 or '10'" );
                break;
        case a == 42;
                console.log( "42 or '42'" );
                break;
        default:
                // 永远执行不到这里
}
// 42 or '42'
```

除简单值以外，case 中还可以出现各种表达式，它会将表达式的结果值和 true 进行比较。因为 a == 42 的结果为 true，所以条件成立。

尽管可以使用 ==，但 switch 中 true 和 true 之间仍然是严格相等比较。即如果 case 表达式的结果为真值，但不是严格意义上的 true（参见第 4 章），则条件不成立。所以，在这里使用 || 和 && 等逻辑运算符就很容易掉进坑里：

```
var a = "hello world";
var b = 10;

switch (true) {
        case (a || b == 10):
                // 永远执行不到这里
                break;
        default:
                console.log( "Oops" );
}
// Oops
```

因为 (a || b == 10) 的结果是 "hello world" 而非 true，所以严格相等比较不成立。此时可以通过强制表达式返回 true 或 false，如 case !!(a || b == 10):（参见第 4 章）。

最后，default 是可选的，并非必不可少（虽然惯例如此）。break 相关规则对 default 仍然适用：

```
var a = 10;

switch (a) {
        case 1:
        case 2:
                // 永远执行不到这里
        default:
                console.log( "default" );
        case 3:
                console.log( "3" );
                break;
        case 4:
                console.log( "4" );
}
// default
// 3
```

正如之前介绍的，case 中的 break 也可以带标签。

上例中的代码是这样执行的，首先遍历并找到所有匹配的 case，如果没有匹配则执行

default 中的代码。因为其中没有 break，所以继续执行已经遍历过的 case 3 代码块，直到 break 为止。

理论上来说，这种情况在 JavaScript 中是可能出现的，但在实际情况中，开发人员一般不会这样来编码。如果确实需要这样做，就应该仔细斟酌并做好注释。

5.8 小结

JavaScript 语法规则中的许多细节需要我们多花点时间和精力来了解。从长远来看，这有助于更深入地掌握这门语言。

语句和表达式在英语中都能找到类比——语句就像英语中的句子，而表达式就像短语。表达式可以是简单独立的，否则可能会产生副作用。

JavaScript 语法规则之上是语义规则（也称作上下文）。例如，{ } 在不同情况下的意思不尽相同，可以是语句块、对象常量、解构赋值（ES6）或者命名函数参数（ES6）。

JavaScript 详细定义了运算符的优先级（运算符执行的先后顺序）和关联（多个运算符的组合方式）。只要熟练掌握了这些规则，就能对如何合理地运用它们作出自己的判断。

ASI（自动分号插入）是 JavaScript 引擎的代码解析纠错机制，它会在需要的地方自动插入分号来纠正解析错误。问题在于这是否意味着大多数的分号都不是必要的（可以省略），或者由于分号缺失导致的错误是否都可以交给 JavaScript 引擎来处理。

JavaScript 中有很多错误类型，分为两大类：早期错误（编译时错误，无法被捕获）和运行时错误（可以通过 try..catch 来捕获）。所有语法错误都是早期错误，程序有语法错误则无法运行。

函数参数和命名参数之间的关系非常微妙。尤其是 arguments 数组，它的抽象泄漏给我们挖了不少坑。因此，尽量不要使用 arguments，如果非用不可，也切勿同时使用 arguments 和其对应的命名参数。

finally 中代码的处理顺序需要特别注意。它们有时能派上很大用场，但也容易引起困惑，特别是在和带标签的代码块混用时。总之，使用 finally 旨在让代码更加简洁易读，切忌弄巧成拙。

switch 相对于 if..else if.. 来说更为简洁。需要注意的一点是，如果对其理解得不够透彻，稍不注意就很容易出错。

附录 A

混合环境 JavaScript

除了本部分之前介绍过的核心的语言机制，JavaScript 程序在实际运行中可能还会出现一些差异。如果 JavaScript 程序仅仅是在引擎中运行的话，它会严格遵循规范并且是可以预测的。但是 JavaScript 程序几乎总是在宿主环境中运行，这使得它在一定程度上变得不可预测。

例如，当你的代码和其他第三方代码一起运行，或者当你的代码在不同的 JavaScript 引擎（并非仅仅是浏览器）上运行时，有些地方就会出现差异。

下面将就此进行简单的介绍。

A.1 Annex B（ECMAScript）

JavaScript 语言的官方名称是 ECMAScript（指的是管理它的 ECMA 标准），这一点不太为人所知。那么 JavaScript 又是指什么呢？ JavaScript 是该语言的通用称谓，更确切地说，它是该规范在浏览器上的实现。

官方 ECMAScript 规范包括 Annex B，其中介绍了由于浏览器兼容性问题导致的与官方规范的差异。

可以这样来理解：这些差异只存在于浏览器中。如果代码只在浏览器中运行，就不会发现任何差异。否则（如果代码也在 Node.js、Rhino 等环境中运行），或者你也不确定的时候，就需要小心对待。

下面是主要的兼容性差异。

- 在非严格模式中允许八进制数值常量存在，如 `0123`（即十进制的 `83`）。
- `window.escape(..)` 和 `window.unescape(..)` 让你能够转义（escape）和回转（unescape）带有 `%` 分隔符的十六进制字符串。例如，`window.escape( "? foo=97%&bar=3%" )` 的结果为 `"%3Ffoo%3D97%25%26bar%3D3%25"`。
- `String.prototype.substr` 和 `String.prototype.substring` 十分相似，除了前者的第二个参数是结束位置索引（非自包含），后者的第二个参数是长度（需要包含的字符数）。

Web ECMAScript

Web ECMAScript 规范（https://javascript.spec.whatwg.org）中介绍了官方 ECMAScript 规范和目前基于浏览器的 JavaScript 实现之间的差异。

换句话说，其中的内容对浏览器来说是“必需的”（考虑到兼容性），但是并未包含在官方规范的“Annex B”部分（到本书写作时）。

- `<!--` 和 `-->` 是合法的单行注释分隔符。
- `String.prototype` 中返回 HTML 格式字符串的附加方法：`anchor(..)`、`big(..)`、`blink(..)`、`bold(..)`、`fixed(..)`、`fontcolor(..)`、`fontsize(..)`、`italics(..)`、`link(..)`、`small(..)`、`strike(..)` 和 `sub(..)`。

以上内容在实际开发中很少使用，也不推荐，我们更倾向于使用其他的内建 DOM API 和自定义工具集。

- `RegExp` 扩展：`RegExp.$1 .. RegExp.$9`（匹配组）和 `RegExp.lastMatch/RegExp["$&"]`（最近匹配）。
- `Function.prototype` 附加方法：`Function.prototype.arguments`（别名为 `arguments` 对象）和 `Function.caller`（别名为 `arguments.caller`）。

`arguments` 和 `arguments.caller` 均已被废止，所以尽量不使用它们，也不要使用它们的别名。

一些十分细微且很不常见的差异这里就不介绍了。如有需要，可参考文档“Annex B”和“Web ECMAScript”。

通常来说，出现这些差异的情况很少，所以无需特别担心。只要在使用它们的时候特别注意即可。

A.2 宿主对象

JavaScript 中有关变量的规则定义得十分清楚，但也不乏一些例外情况，比如自动定义的变量，以及由宿主环境（浏览器等）创建并提供给 JavaScript 引擎的变量——所谓的“宿主对象”（包括内建对象和函数）。

例如：

```
var a = document.createElement( "div" );

typeof a;                               // "object"--正如所料
Object.prototype.toString.call( a ); // "[object HTMLDivElement]"

a.tagName;                              // "DIV"
```

上例中，a 不仅仅是一个 object，还是一个特殊的宿主对象，因为它是一个 DOM 元素。其内部的 [[Class]] 值（为 "HTMLDivElement"）来自预定义的属性（通常也是不可更改的）。

另外一个难点在 4.2.3 节中的“假值对象”部分曾介绍过：一些对象在强制转换为 boolean 时，会意外地成为假值而非真值，这很让人抓狂。

其他需要注意的宿主对象的行为差异有：

- 无法访问正常的 object 内建方法，如 toString();
- 无法写覆盖；
- 包含一些预定义的只读属性；
- 包含无法将 this 重载为其他对象的方法；
- 其他……

在针对运行环境进行编码时，宿主对象扮演着一个十分关键的角色，但要特别注意其行为特性，因为它们常常有别于普通的 JavaScript object。

在我们经常打交道的宿主对象中，console 及其各种方法（log(..)、error(..) 等）是比较值得一提的。console 对象由宿主环境提供，以便从代码中输出各种值。

console 在浏览器中是输出到开发工具控制台，而在 Node.js 和其他服务器端 JavaScript 环境中，则是指向 JavaScript 环境系统进程的标准输出（stdout）和标准错误输出（stderr）。

A.3 全局 DOM 变量

你可能已经知道，声明一个全局变量（使用 var 或者不使用）的结果并不仅仅是创建一个全局变量，而且还会在 global 对象（在浏览器中为 window）中创建一个同名属性。

还有一个不太为人所知的事实是：由于浏览器演进的历史遗留问题，在创建带有 id 属性的 DOM 元素时也会创建同名的全局变量。例如：

```
<div id="foo"></div>
```

以及：

```
if (typeof foo == "undefined") {
    foo = 42;        // 永远也不会运行
}

console.log( foo );  // HTML元素
```

你可能认为只有 JavaScript 代码才能创建全局变量，并且习惯使用 typeof 或 .. in window 来检测全局变量。但是如上例所示，HTML 页面中的内容也会产生全局变量，并且稍不注意就很容易让全局变量检查错误百出。

这也是尽量不要使用全局变量的一个原因。如果确实要用，也要确保变量名的唯一性，从而避免与其他地方的变量产生冲突，包括 HTML 和其他第三方代码。

A.4 原生原型

一个广为人知的 JavaScript 的最佳实践是：不要扩展原生原型。

如果向 Array.prototype 中加入新的方法和属性，假设它们确实有用，设计和命名都很得当，那它最后很有可能会被加入到 JavaScript 规范当中。这样一来你所做的扩展就会与之冲突。

我自己就曾遇到过这样一个例子。

当时我正在为一些网站开发一个嵌入式构件，该构件基于 jQuery（基本上所有的框架都会犯这样的错误）。基本上它在所有的网站上都可以运行，但是在某个网站上却彻底无法运行。

经过差不多一个星期的分析调试之后，我发现这个网站有一段遗留代码，如下：

```
// Netscape 4没有Array.push
Array.prototype.push = function(item) {
    this[this.length-1] = item;
};
```

除了注释以外（谁还会关心 Netscape 4 呢？），上述代码似乎没有问题，是吧？

问题在于 `Array.prototype.push` 随后被加入到了规范中，并且和这段代码不兼容。标准的 `push(..)` 可以一次加入多个值。而这段代码中的 `push` 方法则只会处理第一个值。

几乎所有 JavaScript 框架的代码都使用 `push(..)` 来处理多个值。我的问题则是 CSS 选择器引擎（CSS selector）。可想而知其他很多地方也会有这样的问题。

最初编写这个方法的开发人员将其命名为 `push` 没有问题，但是并未预见到需要处理多个值的情况。这相当于挖了一个坑，而大约 10 年之后，我无意间掉了进去。

从中我们可以吸取几个教训。

首先，不要扩展原生方法，除非你确信代码在运行环境中不会有冲突。如果对此你并非 100% 确定，那么进行扩展是非常危险的。这需要你自己仔细权衡利弊。

其次，在扩展原生方法时需要加入判断条件（因为你可能无意中覆盖了原来的方法）。对于前面的例子，下面的处理方式要更好一些：

```
if (!Array.prototype.push) {
    // Netscape 4没有Array.push
    Array.prototype.push = function(item) {
        this[this.length-1] = item;
    };
}
```

其中，`if` 语句用来确保当 JavaScript 运行环境中没有 `push()` 方法时才将扩展加入。这应该可以解决我的问题。但它并非万全之策，并且存在着一定的隐患。

如果网站代码中的 `push(..)` 原本就不打算处理多个值的情况，那么标准的 `push(..)` 出台后会导致代码运行出错。

如果在 `if` 判断前引入了其他第三方的 `push(..)` 方法，并且该方法的功能不同，也会导致代码运行出错。

这里突出了一个不太为 JavaScript 开发人员注意的问题：在各种第三方代码混合运行的环境中，是否应该只使用现有的原生方法？

答案是否定的，但是实际上不太行得通。通常你无法重新定义所有会用到的原生方法，同时确保它们的安全。即使可以，这种做法也是一种浪费。

那么是否应该既检测原生方法是否存在，又要测试它能否执行我们想要的功能？如果测试没通过，是不是意味着代码要停止执行？

```
// 不要信任 Array.prototype.push
(function(){
```

```
        if (Array.prototype.push) {
            var a = [];
            a.push(1,2);
            if (a[0] === 1 && a[1] === 2) {
                // 测试通过,可以放心使用!
                return;
            }
        }

        throw Error(
            "Array#push() is missing/broken!"
        );
    })();
```

理论上说这个方法不错，但实际上不可能为每个原生函数都做这样的测试。

那应该怎么办呢？我们是否应该逐一做测试？还是假设一切没问题，等出现问题时再处理？

这里没有标准答案。实际上，只要我们自己不去扩展原生原型，就不会遇到这类问题。

如果你和第三方代码都遵循以上原则，那么你的程序是安全的。否则就要更加谨慎小心地对待程序，以防任何可能出现的类似问题。

针对各种运行环境做单元和回归测试能够早点发现问题，却不能够完全杜绝问题。

shim/polyfill

通常来说，在老版本的（不符合规范的）运行环境中扩展原生方法是唯一安全的，因为环境不太可能发生变化——支持新规范的新版本浏览器会完全替代老版本浏览器，而非在老版本上做扩展。

如果能够预见哪些方法会在将来成为新的标准，如 `Array.prototype.foobar`，那么就可以完全放心地使用当前的扩展版本，不是吗？

```
if (!Array.prototype.foobar) {
    // 幼稚
    Array.prototype.foobar = function() {
        this.push( "foo", "bar" );
    };
}
```

如果规范中已经定义了 `Array.prototype.foobar`，并且其功能和上面的代码类似，那就没有什么问题。这种情况一般称为 polyfill（或者 shim）。

polyfill 能有效地为不符合最新规范的老版本浏览器填补缺失的功能，让你能够通过可靠的代码来支持所有你想要支持的运行环境。

ES5-Shim（https://github.com/es-shims/es5-shim）是一个完整的 shim/polyfill 集合，能够为你的项目提供 ES5 基本规范支持。同样，ES6-Shim（https://github.com/es-shims/es6-shim）提供了对 ES6 基本规范的支持。虽然我们可以通过 shim/polyfill 来填补新的 API，但是无法填补新的语法。可以使用 Traceur（https://github.com/google/traceur-compiler/wiki/GettingStarted）这样的工具来实现新旧语法之间的转换。

对于将来可能成为标准的功能，按照大部分人赞同的方式来预先实现能和将来的标准兼容的 polyfill，我们称为 prollyfill（probably fill）。

真正的问题在于一些标准功能无法被完整地 polyfill/prollyfill。

JavaScript 社区存在这样的争论，即是否可以对一个功能做不完整的 polyfill（将无法 polyfill 的部分文档化），或者不做则已，要做就要达到 100% 符合规范。

很多开发人员可以接受一些不完整的 polyfill（如 `Object.create(..)`），因为缺失的部分也不会被用到。

一些人认为在 polyfill/shim 中的 `if` 判断里需要加入兼容性测试，并且只在被测试的功能不存在或者未通过测试时才将其替换。这也是区别 shim（有兼容性测试）和 polyfill（检查功能是否存在）的方式。

对此并没有一个绝对正确的答案。即便在老版本的运行环境中使用了“安全”的做法，对原生功能进行扩展也无法做到 100% 安全。依赖第三方代码中的原生功能也是如此，因为这些功能有可能被扩展了。

因此，在处理这些情况的时候需要格外小心，要编写健壮的代码，并且写好文档。

A.5 <script>

绝大部分网站 /Web 应用程序的代码都存放在多个文件中，通常可以在网页中使用 `<scriopt src=..></script>` 来加载这些文件，或者使用 `<script> .. </script>` 来包含内联代码（inline-code）。

这些文件和内联代码是相互独立的 JavaScript 程序还是一个整体呢？

答案（也许会令人惊讶）是它们的运行方式更像是相互独立的 JavaScript 程序，但是并非总是如此。

它们共享 `global` 对象（在浏览器中则是 `window`），也就是说这些文件中的代码在共享的命名空间中运行，并相互交互。