```
{
    foo: bar()
}
```

很多开发人员以为这里的 { .. } 只是一个孤立的对象常量，没有赋值。事实上不是这样。

{ .. } 在这里只是一个普通的代码块。JavaScript 中这种情况并不多见（在其他语言中则常见得多），但语法上是完全合法的，特别是和 let（块作用域声明）在一起时非常有用（参见《你不知道的 JavaScript（上卷）》的“作用域和闭包”部分）。

{ .. } 和 for/while 循环以及 if 条件语句中代码块的作用基本相同。

但 foo: bar() 这样奇怪的语法为什么也合法呢？

这里涉及 JavaScript 中一个不太为人知（也不建议使用）的特性，叫作“标签语句”（labeled statement）。foo 是语句 bar() 的标签（后面没有 ;，参见 5.3 节）。标签语句具体是做什么用的呢？

如果 JavaScript 有 goto 语句，理论上我们可以使用 goto foo 跳转到 foo 处执行。goto 被公认为是一种极为糟糕的编码方式，它会让代码变得晦涩难懂（也叫作 spaghetti code），好在 JavaScript 不支持 goto。

然而 JavaScript 通过标签跳转能够实现 goto 的部分功能。continue 和 break 语句都可以带一个标签，因此能够像 goto 那样进行跳转。例如：

```
// 标签为foo的循环
foo: for (var i=0; i<4; i++) {
    for (var j=0; j<4; j++) {
        // 如果j和i相等,继续外层循环
        if (j == i) {
            // 跳转到foo的下一个循环
            continue foo;
        }

        // 跳过奇数结果
        if ((j * i) % 2 == 1) {
            // 继续内层循环(没有标签的)
            continue;
        }

        console.log( i, j );
    }
}
// 1 0
// 2 0
// 2 1
// 3 0
// 3 2
```

contine foo 并不是指“跳转到标签 foo 所在位置继续执行”，而是“执行 foo 循环的下一轮循环”。所以这里的 foo 并非 goto。

上例中 continue 跳过了循环 3 1，continue foo（带标签的循环跳转，labeled-loop jump）跳过了循环 1 1 和 2 2。

带标签的循环跳转一个更大的用处在于，和 break __ 一起使用可以实现从内层循环跳转到外层循环。没有它们的话实现起来有时会非常麻烦：

```
// 标签为foo的循环
foo: for (var i=0; i<4; i++) {
    for (var j=0; j<4; j++) {
        if ((i * j) >= 3) {
            console.log( "stopping!", i, j );
            break foo;
        }

        console.log( i, j );
    }
}
// 0 0
// 0 1
// 0 2
// 0 3
// 1 0
// 1 1
// 1 2
// 停止！ 1 3
```

break foo 不是指“跳转到标签 foo 所在位置继续执行”，而是“跳出标签 foo 所在的循环 / 代码块，继续执行后面的代码”。因此它并非传统意义上的 goto。

上例中如果使用不带标签的 break，就可能需要用到一两个函数调用和共享作用域的变量等，这样代码会更难懂，使用带标签的 break 可能更好一些。

标签也能用于非循环代码块，但只有 break 才可以。我们可以对带标签的代码块使用 break ___，但是不能对带标签的非循环代码块使用 continue ___，也不能对不带标签的代码块使用 break：

```
// 标签为bar的代码块
function foo() {
        bar: {
                console.log( "Hello" );
```

```
            break bar;
            console.log( "never runs" );
        }
        console.log( "World" );
}

foo();
// Hello
// World
```

带标签的循环 / 代码块十分少见，也不建议使用。例如，循环跳转也可以通过函数调用来实现。不过在某些情况下它们也能派上用场，这时请务必将注释写清楚！

JSON 被普遍认为是 JavaScript 语言的一个真子集，`{"a":42}` 这样的 JSON 字符串会被当作合法的 JavaScript 代码（请注意 JSON 属性名必须使用双引号！）。其实不是！如果在控制台中输入 `{"a":42}` 会报错。

因为标签不允许使用双引号，所以 `"a"` 并不是一个合法的标签，因此后面不能带 `:`。

JSON 的确是 JavaScript 语法的一个子集，但是 JSON 本身并不是合法的 JavaScript 语法。

这里存在一个十分常见的误区，即如果通过 `<script src=..>` 标签加载 JavaScript 文件，其中只包含 JSON 数据（比如某个 API 返回的结果），那它就会被当作合法的 JavaScript 代码来解析，只不过其内容无法被程序代码访问到。JSON-P（将 JSON 数据封装为函数调用，比如 `foo({"a":42})`）通过将 JSON 数据传递给函数来实现对其的访问。

`{"a":42}` 作为 JSON 值没有任何问题，但是在作为代码执行时会产生错误，因为它会被当作一个带有非法标签的语句块来执行。`foo({"a":42})` 就没有这个问题，因为 `{"a":42}` 在这里是一个传递给 `foo(..)` 的对象常量。所以准确地说，JSON-P 能将 JSON 转换为合法的 JavaScript 语法。

2. 代码块

还有一个坑常被提到（涉及强制类型转换，参见第 4 章）：

```
[] + {}; // "[object Object]"
{} + []; // 0
```

表面上看 + 运算符根据第一个操作数（`[]` 或 `{}`）的不同会产生不同的结果，实则不然。

第一行代码中，`{}` 出现在 + 运算符表达式中，因此它被当作一个值（空对象）来处理。第 4 章讲过 `[]` 会被强制类型转换为 `""`，而 `{}` 会被强制类型转换为 `"[object Object]"`。

但在第二行代码中，`{}` 被当作一个独立的空代码块（不执行任何操作）。代码块结尾不需要分号，所以这里不存在语法上的问题。最后 `+ []` 将 `[]` 显式强制类型转换（参见第 4 章）为 `0`。

3. 对象解构

从 ES6 开始，`{ .. }` 也可用于“解构赋值”（destructuring assignment，详情请参见本系列的《你不知道的 JavaScript（下卷）》的“ES6 & Beyond”部分），特别是对象的解构。例如：

```
function getData() {
    // ..
    return {
        a: 42,
        b: "foo"
    };
}

var { a, b } = getData();

console.log( a, b ); // 42 "foo"
```

`{ a , b } = ..` 就是 ES6 中的解构赋值，相当于下面的代码：

```
var res = getData();
var a = res.a;
var b = res.b;
```

`{ a, b }` 实际上是 `{ a: a, b: b }` 的简化版本，两者均可，只不过 `{ a, b }` 更简洁。

`{ .. }` 还可以用作函数命名参数（named function argument）的对象解构（object destructuring），方便隐式地用对象属性赋值：

```
function foo({ a, b, c }) {
    // 不再需要这样:
    // var a = obj.a, b = obj.b, c = obj.c
    console.log( a, b, c );
}

foo( {
    c: [1,2,3],
    a: 42,
    b: "foo"
} );    // 42 "foo" [1, 2, 3]
```

在不同的上下文中 `{ .. }` 的作用不尽相同，这也是词法和语法的区别所在。掌握这些细节有助于我们了解 JavaScript 引擎解析代码的方式。

4. `else if` 和可选代码块

很多人误以为 JavaScript 中有 `else if`，因为我们可以这样来写代码：

```
if (a) {
    // ..
}
else if (b) {
    // ..
}
else {
    // ..
}
```

事实上 JavaScript 没有 `else if`，但 `if` 和 `else` 只包含单条语句的时候可以省略代码块的 `{ }`。下面的代码你一定不会陌生：

```
if (a) doSomething( a );
```

很多 JavaScript 代码检查工具建议对单条语句也应该加上 `{ }`，如：

```
if (a) { doSomething( a ); }
```

`else` 也是如此，所以我们经常用到的 `else if` 实际上是这样的：

```
if (a) {
    // ..
}
else {
    if (b) {
        // ..
    }
    else {
        // ..
    }
}
```

`if (b) { .. } else { .. }` 实际上是跟在 `else` 后面的一个单独的语句，所以带不带 `{ }` 都可以。换句话说，`else if` 不符合前面介绍的编码规范，`else` 中是一个单独的 `if` 语句。

`else if` 极为常见，能省掉一层代码缩进，所以很受青睐。但这只是我们自己发明的用法，切勿想当然地认为这些都属于 JavaScript 语法的范畴。

5.2 运算符优先级

第 4 章中介绍过，JavaScript 中的 `&&` 和 `||` 运算符返回它们其中一个操作数的值，而非 `true` 或 `false`。在一个运算符两个操作数的情况下这比较好理解：

```
var a = 42;
var b = "foo";

a && b; // "foo"
a || b; // 42
```

那么两个运算符三个操作数呢？

```
var a = 42;
var b = "foo";
var c = [1,2,3];

a && b || c; // ???
a || b && c; // ???
```

想知道结果就需要了解超过一个运算符时表达式的执行顺序。

这些规则被称为“运算符优先级”（operator precedence）。

估计大多数读者都会认为自己已经掌握了运算符优先级。这里我们秉承本系列丛书的一贯宗旨，将对这个主题进行深入探讨，希望读者从中能有新的收获。

回顾前面的例子：

```
var a = 42, b;
b = ( a++, a );

a;  // 43
b;  // 43
```

如果去掉 ( ) 会出现什么情况？

```
var a = 42, b;
b = a++, a;

a;  // 43
b;  // 42
```

为什么上面两个例子中 b 的值会不一样？

原因是，运算符的优先级比 = 低。所以 b = a++，a 其实可以理解为 (b = a++)，a。前面说过 a++ 有后续副作用（after side effect），所以 b 的值是 ++ 对 a 做递增之前的值 42。

这只是一个简单的例子。请务必记住，用，来连接一系列语句的时候，它的优先级最低，其他操作数的优先级都比它高。

回顾前面的一个例子：

```
if (str && (matches = str.match( /[aeiou]/g ))) {
    // ..
}
```

这里对赋值语句使用 ( ) 是必要的，因为 && 运算符的优先级高于 =，如果没有 ( ) 对其中的表达式进行绑定（bind）的话，就会执行作 (str && matches) = str.match..。这样会出错，由于 (str && matches) 的结果并不是一个变量，而是一个 undefined 值，因此它不能

出现在 = 运算符的左边！

下面再来看一个更复杂的例子（本章后面几节都将用到）：

```
var a = 42;
var b = "foo";
var c = false;

var d = a && b || c ? c || b ? a : c && b : a;

d;      // ??
```

应该没有人会写出这样恐怖的代码，这只是用来举例说明多个运算符串联时可能出现的一些常见问题。

上例的结果是 42，当然只要运行一下代码就能够知道答案，但是弄明白其中的来龙去脉更有意思。

首先我们要搞清楚 (a && b || c) 执行的是 (a && b) || c 还是 a && (b || c)？它们之间有什么区别？

```
(false && true) || true;    // true
false && (true || true);    // false
```

事实证明它们是有区别的，false && true || true 的执行顺序如下：

```
false && true || true;      // true
(false && true) || true;    // true
```

&& 先执行，然后是 ||。

那执行顺序是否就一定是从左到右呢？不妨将运算符颠倒一下看看：

```
true || false && false;     // true

(true || false) && false;   // false
true || (false && false);   // true
```

这说明 && 运算符先于 || 执行，而且执行顺序并非我们所设想的从左到右。原因就在于*运算符优先级*。

每门语言都有自己的运算符优先级。遗憾的是，对 JavaScript 运算符优先级有深入了解的开发人员并不多。

如果我们明白其中的道理，上面的例子就是小菜一碟。不过估计很多读者看到上面几个例子时还是需要细细琢磨一番。

遗憾的是，JavaScript 规范对运算符优先级并没有一个集中的介绍，因此我们需要从语法规则中间逐一了解。下面列出一些常见并且有用的优先级规则，以方便查阅。完整列表请参见 MDN 网站（https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Operator_Precedence）上的“优先级列表”。

5.2.1 短路

第 4 章中的附注栏提到过 `&&` 和 `||` 运算符的“短路”（short circuiting）特性。下面我们将对此进行详细介绍。

对 `&&` 和 `||` 来说，如果从左边的操作数能够得出结果，就可以忽略右边的操作数。我们将这种现象称为“短路”（即执行最短路径）。

以 `a && b` 为例，如果 `a` 是一个假值，足以决定 `&&` 的结果，就没有必要再判断 `b` 的值。同样对于 `a || b`，如果 `a` 是一个真值，也足以决定 `||` 的结果，也就没有必要再判断 `b` 的值。

“短路”很方便，也很常用，如：

```
function doSomething(opts) {
    if (opts && opts.cool) {
        // ..
    }
}
```

`opts && opts.cool` 中的 `opts` 条件判断如同一道安全保护，因为如果 `opts` 未赋值（或者不是一个对象），表达式 `opts.cool` 会出错。通过使用短路特性，`opts` 条件判断未通过时 `opts.cool` 就不会执行，也就不会产生错误！

`||` 运算符也一样：

```
function doSomething(opts) {
    if (opts.cache || primeCache()) {
        // ..
    }
}
```

这里首先判断 `opts.cache` 是否存在，如果是则无需调用 `primeCache()` 函数，这样可以避免执行不必要的代码。

5.2.2 更强的绑定

回顾一下前面多个运算符串联在一起的例子：

```
a && b || c ? c || b ? a : c && b : a
```

其中 ? : 运算符的优先级比 && 和 || 高还是低呢？执行顺序是这样？

```
a && b || (c ? c || (b ? a : c) && b : a)
```

还是这样？

```
(a && b || c) ? (c || b) ? a : (c && b) : a
```

答案是后者。因为 && 运算符的优先级高于 ||，而 || 的优先级又高于 ? :。

因此表达式 (a && b || c) 先于包含它的 ? : 运算符执行。另一种说法是 && 和 || 比 ? : 的绑定更强。反过来，如果 c ? c... 的绑定更强，执行顺序就会变成 a && b || (c ? c..)。

5.2.3 关联

&& 和 || 运算符先于 ? : 执行，那么如果多个相同优先级的运算符同时出现，又该如何处理呢？它们的执行顺序是从左到右还是从右到左？

一般说来，运算符的关联（associativity）不是从左到右就是从右到左，这取决于组合（grouping）是从左开始还是从右开始。

请注意：关联和执行顺序不是一回事。

但它为什么又和执行顺序相关呢？原因是表达式可能会产生副作用，比如函数调用：

```
var a = foo() && bar();
```

这里 foo() 首先执行，它的返回结果决定了 bar() 是否执行。所以如果 bar() 在 foo() 之前执行，整个结果会完全不同。

这里遵循从左到右的顺序（JavaScript 的默认执行顺序），与 && 的关联无关。因为上例中只有一个 && 运算符，所以不涉及组合和关联。

而 a && b && c 这样的表达式就涉及组合（隐式），这意味着 a && b 或 b && c 会先执行。

从技术角度来说，因为 && 运算符是左关联（|| 也是），所以 a && b && c 会被处理为 (a && b) && c。不过右关联 a && (b && c) 的结果也一样。

如果 && 是右关联的话会被处理为 a && (b && c)。但这并不意味着 c 会在 b 之前执行。右关联不是指从右往左执行，而是指从右往左组合。任何时候，不论是组合还是关联，严格的执行顺序都应该是从左到右，a，b，然后 c。

所以，&& 和 || 运算符是不是左关联这个问题本身并不重要，只要对此有一个准确的定义

即可。

但情况并非总是这样。一些运算符在左关联和右关联时的表现截然不同。

比如 ? :（即三元运算符或者条件运算符）：

```
a ? b : c ? d : e;
```

? : 是右关联，它的组合顺序是以下哪一种呢？

- a ? b : (c ? d : e)
- (a ? b : c) ? d : e

答案是 a ? b : (c ? d : e)。和 && 以及 || 运算符不同，右关联在这里会影响返回结果，因为 (a ? b : c) ? d : e 对有些值（并非所有值）的处理方式会有所不同。

举个例子：

```
true ? false : true ? true : true;      // false

true ? false : (true ? true : true);    // false
(true ? false : true) ? true : true;    // true
```

在某些情况下，返回的结果没有区别，但其中却有十分微妙的差别。例如：

```
true ? false : true ? true : false;     // false

true ? false : (true ? true : false);   // false
(true ? false : true) ? true : false;   // false
```

这里返回的结果一样，运算符组合看似没起什么作用。然而实际情况是：

```
var a = true, b = false, c = true, d = true, e = false;

a ? b : (c ? d : e); // false, 执行 a 和 b
(a ? b : c) ? d : e; // false, 执行 a, b 和 e
```

这里我们可以看出，? : 是右关联，并且它的组合方式会影响返回结果。

另一个右关联（组合）的例子是 = 运算符。本章前面介绍过一个串联赋值的例子：

```
var a, b, c;

a = b = c = 42;
```

它首先执行 c = 42，然后是 b = ..，最后是 a = ..。因为是右关联，所以它实际上是这样来处理的：a = (b = (c = 42))。

再看看本章前面那个更为复杂的赋值表达式的例子：

```
var a = 42;
var b = "foo";
var c = false;

var d = a && b || c ? c || b ? a : c && b : a;

d;      // 42
```

掌握了优先级和关联等相关知识之后，就能够根据组合规则将上面的代码分解如下：

```
((a && b) || c) ? ((c || b) ? a : (c && b)) : a
```

也可以通过缩进显式让代码更容易理解：

```
(
  (a && b)
    ||
  c
)
  ?
(
  (c || b)
    ?
  a
    :
  (c && b)
)
  :
a
```

现在来逐一执行。

(1) `(a && b)` 结果为 `"foo"`。
(2) `"foo" || c` 结果为 `"foo"`。
(3) 第一个 `?` 中，`"foo"` 为真值。
(4) `(c || b)` 结果为 `"foo"`。
(5) 第二个 `?` 中，`"foo"` 为真值。
(6) `a` 的值为 `42`。

因此，最后结果为 `42`。

5.2.4 释疑

现在你应该对运算符优先级（和关联）有了更深入的了解，多个运算符串联的代码也不在话下。

但是我们仍然面临着一个重要问题，即是不是理解和遵守了运算符优先级和关联规则就万事大吉了？在必要时是否应该使用 `( )` 来自行控制运算符的组合和执行顺序？

换句话说，尽管这些规则是可以学习和掌握的，但其中也不乏问题和陷阱。如果完全依赖它们来编码，就很容易掉进陷阱。那么是否应该经常使用（）来自行控制运算符的执行而不再依赖系统的自动操作呢？

正如第 4 章中的隐式强制类型转换，这个问题仁者见仁，智者见智。对于两者，大多数人的看法都是：要么完全依赖规则编码，要么完全使用显式和自行控制的方式。

对于这个问题，我并没有一个明确的答案。它们各自的优缺点本书都已予以介绍，希望能有助于你加深理解，从而做出自己的判断。

我认为，针对该问题有个折中之策，即在编写程序时要将两者结合起来，既要依赖运算符优先级 / 关联规则，也要适当使用（）自行控制方式。对第 4 章中的隐式强制类型转换也是如此，我们应该安全合理地运用它们，而非无节制地滥用。

例如，如果 `if (a && b && c) ..` 没问题，我就不会使用 `if ((a && b) && c) ..`，因为这样过于繁琐。

然而，如果需要串联两个 ? : 运算符的话，我就会使用（）来自行控制运算符的组合，让代码更清晰易读。

所以我的建议和第 4 章中一样：如果运算符优先级 / 关联规则能够令代码更为简洁，就使用运算符优先级 / 关联规则；而如果（）有助于提高代码可读性，就使用（）。

5.3 自动分号

有时 JavaScript 会自动为代码行补上缺失的分号，即自动分号插入（Automatic Semicolon Insertion，ASI）。

因为如果缺失了必要的 ;，代码将无法运行，语言的容错性也会降低。ASI 能让我们忽略那些不必要的 ;。

请注意，ASI 只在换行符处起作用，而不会在代码行的中间插入分号。

如果 JavaScript 解析器发现代码行可能因为缺失分号而导致错误，那么它就会自动补上分号。并且，只有在代码行末尾与换行符之间除了空格和注释之外没有别的内容时，它才会这样做。

例如：

```
var a = 42, b
c;
```

如果 b 和 c 之间出现 a ，的话（即使另起一行），c 会被作为 var 语句的一部分来处理。在

上例中，JavaScript 判断 b 之后应该有 ;，所以 c; 被处理为一个独立的表达式语句。

又比如：

```
var a = 42, b = "foo";

a
b   // "foo"
```

上述代码同样合法，不会产生错误，因为 ASI 也适用于表达式语句。

ASI 在某些情况下很有用，比如：

```
var a = 42;

do {
    // ..
} while (a) // <-- 这里应该有;
a;
```

语法规定 do..while 循环后面必须带 ;，而 while 和 for 循环后则不需要。大多数开发人员都不记得这一点，此时 ASI 就会自动补上分号。

本章前面讲过，语句代码块结尾不用带 ;，所以不需要用到 ASI：

```
var a = 42;

while (a) {
    // ..
} // <-- 这里可以没有;
a;
```

其他涉及 ASI 的情况是 break、continue、return 和 yield（ES6）等关键字：

```
function foo(a) {
    if (!a) return
    a *= 2;
    // ..
}
```

由于 ASI 会在 return 后面自动加上 ;，所以这里 return 语句并不包括第二行的 a *= 2。return 语句的跨度可以是多行，但是其后必须有换行符以外的代码：

```
function foo(a) {
    return (
        a * 2 + 3 / 12
    );
}
```

上述规则对 break、continue 和 yield 也同样适用。

纠错机制

是否应该完全依赖 ASI 来编码，这是 JavaScript 社区中最具争议性的话题之一（除此之外还有 Tab 和空格之争）。

大多数情况下，分号并非必不可少，不过 `for( .. ) ..` 循环头部的两个分号是必需的。

正方认为 ASI 机制大有裨益，能省略掉那些不必要的 `;`，让代码更简洁。此外，ASI 让许多 `;` 变得可有可无，因此只要代码没问题，有没有 `;` 都一样。

反方则认为 ASI 机制问题太多，对于缺乏经验的初学者尤其如此，因为自动插入 `;` 会无意中改变代码的逻辑。还有一些开发人员认为省略分号本身就是错误的，应该通过 linter 这样的工具来找出这些错误，而不是依赖 JavaScript 引擎来改正错误。

仔细阅读规范就会发现，ASI 实际上是一个“纠错”（error correction）机制。这里的错误是指*解析器错误*。换句话说，ASI 的目的在于提高解析器的容错性。

究竟哪些情况需要容错呢？我认为，解析器报错就意味着代码有问题。对 ASI 来说，解析器报错的唯一原因就是代码中缺失了必要的分号。

我认为在代码中省略那些“不必要的分号”就意味着“这些代码解析器无法解析，但是仍然可以运行”。

仅仅为了追求“代码的美观”，省去一些键盘输入，这样做不免有点得不偿失。

这与空格和 Tab 之争还不是一回事，后者仅涉及代码的美观问题，前者则关系到原则问题：是遵循语法规则来编码，还是打规则的擦边球。

换个角度来看，依赖于 ASI 实际上是将换行符当作有意义的“空格”来对待。在一些语言（如 Python）中空格是有意义的，但这对 JavaScript 是否适用呢？

我建议在所有需要的地方加上分号，将对 ASI 的依赖降到最低。

以上观点并非一家之言。JavaScript 的作者 Brendan Eich 早在 2012 年就说过这样的话（http://brendaneich.com/2012/04/the-infernal-semicolon/）：

> ASI 是一个语法纠错机制。若将换行符当作有意义的字符来对待，就会遇到很多问题。多希望在 1995 年 5 月的那十天里（ECMAScript 规范制定期间），我让换行符承载了更多的意义。但切勿认为 ASI 真的会将换行符当作有意义的字符。

5.4 错误

JavaScript 不仅有各种类型的运行时错误（`TypeError`、`ReferenceError`、`SyntaxError` 等），

它的语法中也定义了一些编译时错误。

在编译阶段发现的代码错误叫作“早期错误”（early error）。语法错误是早期错误的一种（如 `a = ,`）。另外，语法正确但不符合语法规则的情况也存在。

这些错误在代码执行之前是无法用 `try..catch` 来捕获的，相反，它们还会导致解析 / 编译失败。

规范没有明确规定浏览器（和开发工具）应该如何处理报错，因此下面的报错处理（包括错误类型和错误信息）在不同的浏览器中可能会有所不同。

举个简单的例子：正则表达式常量中的语法。这里 JavaScript 语法没有问题，但非法的正则表达式也会产生早期错误：

```
var a = /+foo/;      // 错误!
```

语法规定赋值对象必须是一个标识符（identifier，或者 ES6 中的解构表达式），因此下面的 42 会报错：

```
var a;
42 = a;      // 错误!
```

ES5 规范的严格模式定义了很多早期错误。比如在严格模式中，函数的参数不能重名：

```
function foo(a,b,a) { }                 // 没问题

function bar(a,b,a) { "use strict"; }   // 错误!
```

再如，对象常量不能包含多个同名属性：

```
(function(){
    "use strict";

    var a = {
        b: 42,
        b: 43
    };          // 错误!
})();
```

从语义角度来说，这些错误并非*词法*错误，而是*语法*错误，因为它们在词法上是正确的。只不过由于没有 `GrammarError` 类型，一些浏览器选择用 `SyntaxError` 来代替。