的原因，旧有的八进制 052 形式在非严格模式下还是合法的（尽管没有指出），但不应该再使用这种形式了。

下面是新的 ES6 数字字面量形式：

```
var dec = 42,
    oct = 0o52,          // 或者0O52 :(
    hex = 0x2a,          // 或者0X2a :/
    bin = 0b101010;      // 或者0B101010 :/
```

唯一合法的小数形式是十进制的。八进制、十六进制和二进制都是整数形式。

这些形式的字符串表示都可以强制类型转换 / 变换成相应的数字值：

```
Number( "42" );          // 42
Number( "0o52" );        // 42
Number( "0x2a" );        // 42
Number( "0b101010" );    // 42
```

尽管并非是 ES6 中全新的，但有一个不为人知的事实就是这些形式都可以进行（某种程度的）反向转换：

```
var a = 42;

a.toString();            // "42"--也可以用a.toString( 10 )
a.toString( 8 );         // "52"
a.toString( 16 );        // "2a"
a.toString( 2 );         // "101010"
```

实际上，可以用这种方式以任何 2 到 36 之间的基表示一个数字，虽然像 2、8、10 和 16 这些标准基范围之外的表示法非常少见。

## 2.12 Unicode

先声明这一小节并非是对 Unicode 资源完整详尽的介绍。我将会覆盖 ES6 中你需要了解的关于 Unicdoe 的**变化**，但也不会比这更深入太多了。关于 JavaScript 和 Unicode，Mathias Bynens（http://twitter.com/ mathias）编写 / 发表了详尽睿智的介绍（参考 https://mathiasbynens.be/notes/javascript-unicode 和 http:// fluentconf.com/javascript-html-2015/public/content/ 2015/02/18-javascript-loves-unicode）。

Unicode 字符范围从 0x0000 到 0xFFFF，包含可能看到和接触到的所有（各种语言的）标准打印字符。这组字符称为**基本多语言平面**（Basic Multilingual Plane，BMP）。BMP 甚至包含了像雪人这样的有趣的符号：☃（U+2603）。

在 BMP 集之外还有很多其他扩展 Unicode 字符，范围直到 0x10FFFF。这些符号通常是**星形符号**（astral symbol），这个名称是指 BMP 之外的字符的 16 个**平面**（或者说，层次 / 分

组）的集合。星形符号的例子包括𝄞 U+1D11E）和💩 U+1F4A9) 这样的符号。

在 ES6 之前，可以通过 Unicode 转义符指定 JavaScript 字符串为 Unicode 字符，比如：

```
var snowman = "\u2603";
console.log( snowman );       // "☃"
```

但是，Unicode 转义符 `\uXXXX` 只支持四个十六进制字符的形式，所以这种方法只能表示 BMP 字符集。要在 ES6 之前用 Unicode 转义符表示 astral 字符，需要使用一个**替代对**——简单说就是连续两个特别计算出来的 Unicode 转义字符，JavaScript 解释器会把它解释为单个 astral 字符：

```
var gclef = "\uD834\uDD1E";
console.log( gclef );        // "𝄞"
```

而在 ES6 中，现在有了可以用于作 Unicode 转义（在字符串和正则表达式中）的新形式，称为 Unicode **码点转义**（code point escaping）：

```
var gclef = "\u{1D11E}";
console.log( gclef );        // "𝄞"
```

你可以看到，区别在于转义序列中出现了 `{ }`，支持在其中包含任意多个十六进制字符。因为最多只需要六个就可以表示 Unicode 中最大的可能码点（也就是 `0x10FFFF`），所以这足够了。

## 2.12.1 支持 Unicode 的字符串运算

默认情况下，JavaScript 字符串运算和方法不能感知字符串中的 astral 符号。所以会单独处理每个 BMP 字符，即使是构成单个 astral 字符的两半。考虑：

```
var snowman = "☃";
snowman.length;                  // 1

var gclef = "𝄞";
gclef.length;                    // 2
```

那么如何精确计算这样的字符串的长度呢？在这种情况下，可以使用下面的技巧：

```
var gclef = "𝄞";

[...gclef].length;               // 1
Array.from( gclef ).length;      // 1
```

回忆本章前面 2.9 节所介绍的，ES6 字符串有内建的迭代器。这个迭代器恰好是可以识别 Unicode 的，也就是说它能够自动将 astral 符号作为单个值输出。我们可以利用这一点，在数组字面量使用 ...spread 运算符，创建一个字符串符号的数组。然后查看结果数组的长

度。ES6 的 `Array.from(..)` 所做的事情基本上和 `[...XYZ]` 一样，我们将在第 6 章详细介绍这个工具。

应该注意，与理论上优化过的原生工具 / 属性的做法相比，只是为了得到一个字符串的长度就需要构造并消耗一个迭代器，这种方法的性能代价相对来说是非常昂贵的。

不幸的是，完整的答案并没有那么简单直接。除了替代字符对（字符串迭代器会处理），还有特殊 Unicode 码点需要用特殊的方式处理，这就更难计算了。例如，有一组码点会修改前面相邻的字符，被称为**组合音标符号**（Combining Diacritical Mark)。

考虑这两个字符串输出：

```
console.log( s1 );              // "é"
console.log( s2 );              // "é"
```

看起来是一样的，但实际上并非如此！下面是创建 `s1` 和 `s2` 的过程：

```
var s1 = "\xE9",
    s2 = "e\u0301";
```

可能你已经猜到，前面的 `length` 计算技巧对于 `s2` 并不适用：

```
[...s1].length;                 // 1
[...s2].length;                 // 2
```

那么应该怎么办呢？这种情况下，可以在查询长度之前使用 ES6 的 `String#normalize(..)` 工具（我们将在第 6 章进一步介绍）对这个值执行 **Unicode 规范化**（Unicode normalization）：

```
var s1 = "\xE9",
    s2 = "e\u0301";

s1.normalize().length;          // 1
s2.normalize().length;          // 1

s1 === s2;                      // false
s1 === s2.normalize();          // true
```

基本上说就是，`normalize(..)` 接受像 `"e\u0301"` 这样的一个序列，然后把它规范化为 `"\xE9"`。甚至如果有合适的 Unicode 符号可以合并的话，规范化可以把多个相邻的组合符号合并：

```
var s1 = "o\u0302\u0300",
    s2 = s1.normalize(),
    s3 = "ồ";

s1.length;                      // 3
```

```
s2.length;                  // 1
s3.length;                  // 1

s2 === s3;                  // true
```

不幸的是，这里规范化也并不完美。如果有多个组合符号修改了单个字符，你可能就无法得到期望的长度结果，因为可能并没有单个的定义好的规范化符号可以表示所有这些符号带来的合并结果。举例来说：

```
var s1 = "e\u0301\u0330";

console.log( s1 );          // "ḗ"

s1.normalize().length;      // 2
```

你越是深入挖掘这个兔子洞，就越会意识到很难得到一个对“长度”的精确定义。我们视觉上看到的渲染出来的单个字符——更精确的叫法是**字素**（grapheme）——并不总是与程序处理意义上的单个“字符”严格对应。

如果你想了解这个兔子洞到底有多深，参见“字素族界限”算法（http://www.Unicode.org/reports/tr29/#Grapheme_Cluster_Boundaries）。

## 2.12.2 字符定位

与长度复杂性类似，“位于位置 2 处的字符是什么？”的精确含义又是什么呢？原生的前 ES6 对此的答案是 `charAt(..)`，但它不支持 astral 字符的原子性，也不会考虑组合符号的因素。

考虑：

```
var s1 = "abc\u0301d",
    s2 = "ab\u0107d",
    s3 = "ab\u{1d49e}d";

console.log( s1 );          // "abćd"
console.log( s2 );          // "abćd"
console.log( s3 );          // "ab𝒞d"

s1.charAt( 2 );             // "c"
s2.charAt( 2 );             // "ć"
s3.charAt( 2 );             // "" <-- 不可打印
s3.charAt( 3 );             // "" <-- 不可打印
```

那么 ES6 是否给出了支持 Unicode 版本的 `charAt(..)` 呢？不幸的是，并没有。但在编写本部分时，已经有一个这样的后 ES6 提案在考虑之中了。

但有了前一小节中介绍的内容（当然也包括给出提醒的局限性！），我们可以给出 ES6 版本的回答：

```
var s1 = "abc\u0301d",
    s2 = "ab\u0107d",
    s3 = "ab\u{1d49e}d";

[...s1.normalize()][2];          // "ć"
[...s2.normalize()][2];          // "ć"
[...s3.normalize()][2];          // "𝒞"
```

记住前面的提醒：从性能的角度看，每次想要得到单个字符都要构造并消耗一个迭代器是……非常不理想的。我们期待后 ES6 优化的内建工具尽快出现。

那么支持 Unicode 的版本工具 `charCodeAt(..)` 怎么样呢？ ES6 提供了 `codePointAt(..)`：

```
var s1 = "abc\u0301d",
    s2 = "ab\u0107d",
    s3 = "ab\u{1d49e}d";

s1.normalize().codePointAt( 2 ).toString( 16 );
// "107"

s2.normalize().codePointAt( 2 ).toString( 16 );
// "107"

s3.normalize().codePointAt( 2 ).toString( 16 );
// "1d49e"
```

反方向呢？ ES6 中支持 Unicode 版本的 `String.fromCharCode(..)` 是 `String.fromCodePoint(..)`：

```
String.fromCodePoint( 0x107 );       // "ć"

String.fromCodePoint( 0x1d49e );     // "𝒞"
```

那么稍等，能不能组合 `String.fromCodePoint(..)` 和 `codePointAt(..)` 来获得支持 Unicode 的 `charAt(..)` 的更简单且更优的方法呢？是的，能！

```
var s1 = "abc\u0301d",
    s2 = "ab\u0107d",
    s3 = "ab\u{1d49e}d";

String.fromCodePoint( s1.normalize().codePointAt( 2 ) );
// "ć"

String.fromCodePoint( s2.normalize().codePointAt( 2 ) );
// "ć"
```

```
String.fromCodePoint( s3.normalize().codePointAt( 2 ) );
// "𝒞"
```

还有很多字符串方法这里没有介绍，包括 `toUpperCase()`、`toLowerCase()`、`substring(..)`、`indexOf(..)`、`slice(..)`，以及几十个其他方法。这些方法都没有修改或增补以提供完整的 Unicode 支持，所以在处理包含 astral 符号的字符串的时候应该非常小心——还是尽可能避免使用吧！

也有一些字符串方法使用了正则表达式实现其功能，比如 `replace(..)` 和 `match(..)`。谢天谢地，正如我们在 2.10.1 节中介绍的，ES6 为正则表达式提供了 Unicode 支持。

好吧，现在可以了！通过前面介绍的各种新增特性可以看出，JavaScript 的 Unicode 字符串支持明显优于前 ES6 版本（尽管还不完美）。

### 2.12.3 Unicode 标识符名

Unicode 也可以用作标识符名（变量、属性等）。在 ES6 之前，可以通过 Unicode 转义符实现这一点，比如：

```
var \u03A9 = 42;

// 等价于：var Ω = 42;
```

而在 ES6 中，还可以使用前面解释过的码点转义符语法：

```
var \u{2B400} = 42;

// 等价于：var 𫐀 = 42;
```

关于到底支持哪些 Unicode 字符，有一套复杂的规则。另外，还有一些只允许出现在标识符名称中的非首个字符位置上。

有关所有这些细节，Mathias Bynens 写了一篇很好的文章（https://mathiasbynens.be/notes/javascript-identifiers-es6）。

在标识符名称中使用这些不常见字符的原因是很罕见的，也只是学术意义上的。通常最好不要编写依赖于这些晦涩特性的代码。

## 2.13 符号

ES6 为 JavaScript 引入了一个新的原生类型：`symbol`，这是很久没有发生过的事情。但是，和其他原生类型不一样，symbol 没有字面量形式。

下面是创建 symbol 的过程：

```
var sym = Symbol( "some optional description" );

typeof sym;     // "symbol"
```

以下几点需要注意。

- 不能也不应该对 `Symbol(..)` 使用 `new`。它并不是一个构造器，也不会创建一个对象。
- 传给 `Symbol(..)` 的参数是可选的。如果传入了的话，应该是一个为这个 symbol 的用途给出用户友好描述的字符串。
- `typeof` 的输出是一个新的值 (`"symbol"`)，这是识别 symbol 的首选方法。

如果提供了描述的话，它只被用作为这个符号的字符串表示：

```
sym.toString();     // "Symbol(some optional description)"
```

如同原生字符串值不是 `String` 的实例一样，symbol 也不是 `Symbol` 的实例。如果出于某种原因想要构造一个 symbol 值的装箱封装对象形式，可以使用下面的方法：

```
sym instanceof Symbol;       // false

var symObj = Object( sym );
symObj instanceof Symbol;    // true

symObj.valueOf() === sym;    // true
```

这段代码中的 `symObj` 也可以换作 `sym`；两种形式都在所有使用 symbol 的场合适用。需要使用装箱封装对象形式（`symObj`）而不是原生形式（`sym`）的情况很少。和针对其他原生类型的建议一样，最好使用 `sym` 代替 `symObj`。

符号本身的内部值——称为它的名称（`name`）——是不在代码中出现且无法获得的。可以把这个符号值想象为一个自动生成的、（在应用内部）唯一的字符串值。

但如果这个值是隐藏的且无法获得，那么符号的存在意义是什么呢？

符号的主要意义是创建一个类 ( 似 ) 字符串的不会与其他任何值冲突的值。所以，考虑使用一个符号作为事件名的常量表示的例子：

```
const EVT_LOGIN = Symbol( "event.login" );
```

然后在需要像 `"event.login"` 这样的一般字符串字面量的地方就可以使用 `EVT_LOGIN` 了：

```
evthub.listen( EVT_LOGIN, function(data){
```

```
    // ..
} );
```

这里的好处是 EVT_LOGIN 持有一个不可能与其他值（有意或无意）重复的值，所以这里分发或处理的事件不会有任何混淆。

在底层，前面代码中的 evthub 工具很可能在某个用来跟踪事件处理函数的内部对象（hash）中直接使用 EVT_LOGIN 参数的符号值属性 / 键值。如果 evthub 需要把这个符号值作为一个真正的字符串使用，那么它会需要用 String() 或者 toString() 进行显式类型转换，因为不允许隐式地把符号转换为字符串。

可以在对象中直接使用符号作为属性名 / 键值，比如用作一个特殊的想要作为隐藏或者元属性的属性。尽管通常会这么使用，但是它**实际上**并不是隐藏的或者无法接触的属性，了解这一点很重要。

考虑一下这个实现了**单例**（singleton）模式的模块，也就是说，它只允许自己被创建一次：

```
const INSTANCE = Symbol( "instance" );

function HappyFace() {

    if (HappyFace[INSTANCE]) return HappyFace[INSTANCE];

    function smile() { .. }

    return HappyFace[INSTANCE] = {
        smile: smile
  };
}

var me = HappyFace(),
    you = HappyFace();

me === you;                     // true
```

这里的 INSTANCE 符号值是一个特殊的、几乎隐藏的、类似元属性的属性，静态保存在 HappeyFace() 函数对象中。

也可以采用平凡的、旧有的属性，比如 __instance，其行为特性是相同的。符号的使用只是改进了元编程风格，把 INSTANCE 属性与其他普通属性区分开来。

## 2.13.1 符号注册

在后几个例子中使用符号有几个细微缺点，EVT_LOGIN 和 INSTANCE 变量不得不保存在外层作用域中（可能甚至是全局作用域），或者保存在某个公开可用的位置，这样所有需要使

用这些符号的代码才都能够访问它们。

要改进访问这些符号的代码的组织形式，可以通过**全局符号注册**（global symbol registry）创建这些符号值。举例来说：

```
const EVT_LOGIN = Symbol.for( "event.login" );

console.log( EVT_LOGIN );       // Symbol(event.login)
```

以及

```
function HappyFace() {
    const INSTANCE = Symbol.for( "instance" );

    if (HappyFace[INSTANCE]) return HappyFace[INSTANCE];

    // ..

    return HappyFace[INSTANCE] = { .. };
}
```

`Symbol.for(..)` 在全局符号注册表中搜索，来查看是否有描述文字相同的符号已经存在，如果有的话就返回它。如果没有的话，会新建一个并将其返回。换句话说，全局注册表把符号值本身根据其描述文字作为单例处理。

但是，这也意味着只要使用的描述名称匹配，可以在应用的任何地方通过 `Symbol.for(..)` 从注册表中获取这个符号。

具有讽刺意义的是，基本上符号的目的是为了取代应用中的 magic **字符串**（magic string，赋予特殊意义的任意字符串）。但在全局符号注册表中恰恰是用 magic 字符串值来唯一标识 / 定位符号。

为了避免意外冲突，可能需要符号描述唯一。一个简单的实现方法是在其中包含前缀 / 上下文 / 名字空间信息。

例如，考虑下面这个工具：

```
function extractValues(str) {
    var key = Symbol.for( "extractValues.parse" ),
        re = extractValues[key] ||
            /[^=&]+?=([^&]+?)(?=&|$)/g,
        values = [], match;

    while (match = re.exec( str )) {
        values.push( match[1] );
}

    return values;
}
```

这里使用了 "extractValues.parse" 这个 magic 字符串值，因为注册表中其他符号的描述与之冲突的可能性不大。

如果这个工具的用户想要覆盖解析正则表达式，也可以使用符号注册：

```
extractValues[Symbol.for( "extractValues.parse" )] =
    /..some pattern../g;

extractValues( "..some string.." );
```

符号注册为这些值提供了全局存储，除了这个帮助之外，这里看到的所有示例实际上都可以通过直接用 magic 字符串 "extractValues.parse"，而不是符号作为键值来实现。其改进更多是在元编程这一层次上而不是在函数这一层。

可以使用已经存储在注册中的符号值寻找其底层存储的描述文本（键值）。比如，因为无法传递符号本身，可能需要向应用的另外一部分发送信号告诉它如何在注册表中定位这个符号。

可以使用 Symbol.keyFor(..) 提取注册符号的描述文本（键值）：

```
var s = Symbol.for( "something cool" );

var desc = Symbol.keyFor( s );
console.log( desc );            // "something cool"

// 再次从注册中取得符号
var s2 = Symbol.for( desc );

s2 === s;                       // true
```

## 2.13.2 作为对象属性的符号

如果把符号用作对象的属性 / 键值，那么它会以一种特殊的方式存储，使得这个属性不出现在对这个对象的一般属性枚举中：

```
var o = {
    foo: 42,
    [ Symbol( "bar" ) ]: "hello world",
    baz: true
};

Object.getOwnPropertyNames( o );    // [ "foo","baz" ]
```

要取得对象的符号属性：

```
Object.getOwnPropertySymbols( o );  // [ Symbol(bar) ]
```