NaN 有几个独特的属性。首先，任何涉及 NaN 的操作始终返回 NaN（如 NaN/10），在连续多步计算时这可能是个问题。其次，NaN 不等于包括 NaN 在内的任何值。例如，下面的比较操作会返回 false：

```
console.log(NaN == NaN); // false
```

为此，ECMAScript 提供了 isNaN()函数。该函数接收一个参数，可以是任意数据类型，然后判断这个参数是否“不是数值”。把一个值传给 isNaN()后，该函数会尝试把它转换为数值。某些非数值的值可以直接转换成数值，如字符串"10"或布尔值。任何不能转换为数值的值都会导致这个函数返回 true。举例如下：

```
console.log(isNaN(NaN));      // true
console.log(isNaN(10));       // false, 10 是数值
console.log(isNaN("10"));     // false, 可以转换为数值 10
console.log(isNaN("blue"));   // true, 不可以转换为数值
console.log(isNaN(true));     // false, 可以转换为数值 1
```

上述的例子测试了 5 个不同的值。首先测试的是 NaN 本身，显然会返回 true。接着测试了数值 10 和字符串"10"，都返回 false，因为它们的数值都是 10。字符串"blue"不能转换为数值，因此函数返回 true。布尔值 true 可以转换为数值 1，因此返回 false。

> **注意**　虽然不常见，但 isNaN()可以用于测试对象。此时，首先会调用对象的 valueOf()方法，然后再确定返回的值是否可以转换为数值。如果不能，再调用 toString()方法，并测试其返回值。这通常是 ECMAScript 内置函数和操作符的工作方式，本章后面会讨论。

4. 数值转换

有 3 个函数可以将非数值转换为数值：Number()、parseInt()和 parseFloat()。Number()是转型函数，可用于任何数据类型。后两个函数主要用于将字符串转换为数值。对于同样的参数，这 3 个函数执行的操作也不同。

Number()函数基于如下规则执行转换。

- 布尔值，true 转换为 1，false 转换为 0。
- 数值，直接返回。
- null，返回 0。
- undefined，返回 NaN。
- 字符串，应用以下规则。
  - 如果字符串包含数值字符，包括数值字符前面带加、减号的情况，则转换为一个十进制数值。因此，Number("1")返回 1，Number("123")返回 123，Number("011")返回 11（忽略前面的零）。
  - 如果字符串包含有效的浮点值格式如"1.1"，则会转换为相应的浮点值（同样，忽略前面的零）。
  - 如果字符串包含有效的十六进制格式如"0xf"，则会转换为与该十六进制值对应的十进制整数值。
  - 如果是空字符串（不包含字符），则返回 0。
  - 如果字符串包含除上述情况之外的其他字符，则返回 NaN。
- 对象，调用 valueOf()方法，并按照上述规则转换返回的值。如果转换结果是 NaN，则调用 toString()方法，再按照转换字符串的规则转换。

从不同数据类型到数值的转换有时候会比较复杂，看一看 Number()的转换规则就知道了。下面是几个具体的例子：

```
let num1 = Number("Hello world!");  // NaN
let num2 = Number("");              // 0
let num3 = Number("000011");        // 11
let num4 = Number(true);            // 1
```

可以看到，字符串"Hello world"转换之后是 NaN，因为它找不到对应的数值。空字符串转换后是 0。字符串 000011 转换后是 11，因为前面的零被忽略了。最后，true 转换为 1。

> **注意** 本章后面会讨论到的一元加操作符与 Number()函数遵循相同的转换规则。

考虑到用 Number()函数转换字符串时相对复杂且有点反常规，通常在需要得到整数时可以优先使用 parseInt()函数。parseInt()函数更专注于字符串是否包含数值模式。字符串最前面的空格会被忽略，从第一个非空格字符开始转换。如果第一个字符不是数值字符、加号或减号，parseInt()立即返回 NaN。这意味着空字符串也会返回 NaN（这一点跟 Number()不一样，它返回 0）。如果第一个字符是数值字符、加号或减号，则继续依次检测每个字符，直到字符串末尾，或碰到非数值字符。比如，"1234blue"会被转换为 1234，因为"blue"会被完全忽略。类似地，"22.5"会被转换为 22，因为小数点不是有效的整数字符。

假设字符串中的第一个字符是数值字符，parseInt()函数也能识别不同的整数格式（十进制、八进制、十六进制）。换句话说，如果字符串以"0x"开头，就会被解释为十六进制整数。如果字符串以"0"开头，且紧跟着数值字符，在非严格模式下会被某些实现解释为八进制整数。

下面几个转换示例有助于理解上述规则：

```
let num1 = parseInt("1234blue");  // 1234
let num2 = parseInt("");          // NaN
let num3 = parseInt("0xA");       // 10，解释为十六进制整数
let num4 = parseInt(22.5);        // 22
let num5 = parseInt("70");        // 70，解释为十进制值
let num6 = parseInt("0xf");       // 15，解释为十六进制整数
```

不同的数值格式很容易混淆，因此 parseInt()也接收第二个参数，用于指定底数（进制数）。如果知道要解析的值是十六进制，那么可以传入 16 作为第二个参数，以便正确解析：

```
let num = parseInt("0xAF", 16); // 175
```

事实上，如果提供了十六进制参数，那么字符串前面的"0x"可以省掉：

```
let num1 = parseInt("AF", 16);  // 175
let num2 = parseInt("AF");      // NaN
```

在这个例子中，第一个转换是正确的，而第二个转换失败了。区别在于第一次传入了进制数作为参数，告诉 parseInt()要解析的是一个十六进制字符串。而第二个转换检测到第一个字符就是非数值字符，随即自动停止并返回 NaN。

通过第二个参数，可以极大扩展转换后获得的结果类型。比如：

```
let num1 = parseInt("10", 2);   // 2，按二进制解析
let num2 = parseInt("10", 8);   // 8，按八进制解析
let num3 = parseInt("10", 10);  // 10，按十进制解析
let num4 = parseInt("10", 16);  // 16，按十六进制解析
```

因为不传底数参数相当于让 parseInt()自己决定如何解析，所以为避免解析出错，建议始终传给它第二个参数。

> **注意**　多数情况下解析的应该都是十进制数，此时第二个参数就要传入 10。

parseFloat()函数的工作方式跟 parseInt()函数类似，都是从位置 0 开始检测每个字符。同样，它也是解析到字符串末尾或者解析到一个无效的浮点数值字符为止。这意味着第一次出现的小数点是有效的，但第二次出现的小数点就无效了，此时字符串的剩余字符都会被忽略。因此，"22.34.5"将转换成 22.34。

parseFloat()函数的另一个不同之处在于，它始终忽略字符串开头的零。这个函数能识别前面讨论的所有浮点格式，以及十进制格式（开头的零始终被忽略）。十六进制数值始终会返回 0。因为 parseFloat()只解析十进制值，因此不能指定底数。最后，如果字符串表示整数（没有小数点或者小数点后面只有一个零），则 parseFloat()返回整数。下面是几个示例：

```
let num1 = parseFloat("1234blue");  // 1234，按整数解析
let num2 = parseFloat("0xA");       // 0
let num3 = parseFloat("22.5");      // 22.5
let num4 = parseFloat("22.34.5");   // 22.34
let num5 = parseFloat("0908.5");    // 908.5
let num6 = parseFloat("3.125e7");   // 31250000
```

3.4.6　String 类型

String（字符串）数据类型表示零或多个 16 位 Unicode 字符序列。字符串可以使用双引号（"）、单引号（'）或反引号（`）标示，因此下面的代码都是合法的：

```
let firstName = "John";
let lastName = 'Jacob';
let lastName = `Jingleheimerschmidt`
```

跟某些语言中使用不同的引号会改变对字符串的解释方式不同，ECMAScript 语法中表示字符串的引号没有区别。不过要注意的是，以某种引号作为字符串开头，必须仍然以该种引号作为字符串结尾。比如，下面的写法会导致语法错误：

```
let firstName = 'Nicholas"; // 语法错误：开头和结尾的引号必须是同一种
```

1. 字符字面量

字符串数据类型包含一些字符字面量，用于表示非打印字符或有其他用途的字符，如下表所示：

字 面 量	含　义
\n	换行
\t	制表
\b	退格
\r	回车
\f	换页
\\	反斜杠（\）
\'	单引号（'），在字符串以单引号标示时使用，例如'He said, \'hey.\''

（续）

字 面 量	含 义
\"	双引号（"），在字符串以双引号标示时使用，例如"He said, \"hey.\""
\`	反引号（`），在字符串以反引号标示时使用，例如`He said, \`hey.\``
\x*nn*	以十六进制编码 *nn* 表示的字符（其中 *n* 是十六进制数字 0~F），例如\x41 等于"A"
\u*nnnn*	以十六进制编码 *nnnn* 表示的 Unicode 字符（其中 *n* 是十六进制数字 0~F），例如\u03a3 等于希腊字符"Σ"

这些字符字面量可以出现在字符串中的任意位置，且可以作为单个字符被解释：

```
let text = "This is the letter sigma: \u03a3.";
```

在这个例子中，即使包含 6 个字符长的转义序列，变量 text 仍然是 28 个字符长。因为转义序列表示一个字符，所以只算一个字符。

字符串的长度可以通过其 length 属性获取：

```
console.log(text.length); // 28
```

这个属性返回字符串中 16 位字符的个数。

> **注意** 如果字符串中包含双字节字符，那么 length 属性返回的值可能不是准确的字符数。第 5 章将具体讨论如何解决这个问题。

2. 字符串的特点

ECMAScript 中的字符串是不可变的（immutable），意思是一旦创建，它们的值就不能变了。要修改某个变量中的字符串值，必须先销毁原始的字符串，然后将包含新值的另一个字符串保存到该变量，如下所示：

```
let lang = "Java";
lang = lang + "Script";
```

这里，变量 lang 一开始包含字符串"Java"。紧接着，lang 被重新定义为包含"Java"和"Script"的组合，也就是"JavaScript"。整个过程首先会分配一个足够容纳 10 个字符的空间，然后填充上"Java"和"Script"。最后销毁原始的字符串"Java"和字符串"Script"，因为这两个字符串都没有用了。所有处理都是在后台发生的，而这也是一些早期的浏览器（如 Firefox 1.0 之前的版本和 IE6.0）在拼接字符串时非常慢的原因。这些浏览器在后来的版本中都有针对性地解决了这个问题。

3. 转换为字符串

有两种方式把一个值转换为字符串。首先是使用几乎所有值都有的 toString()方法。这个方法唯一的用途就是返回当前值的字符串等价物。比如：

```
let age = 11;
let ageAsString = age.toString();       // 字符串"11"
let found = true;
let foundAsString = found.toString();   // 字符串"true"
```

toString()方法可见于数值、布尔值、对象和字符串值。（没错，字符串值也有 toString()方法，该方法只是简单地返回自身的一个副本。）null 和 undefined 值没有 toString()方法。

多数情况下，toString()不接收任何参数。不过，在对数值调用这个方法时，toString()可以

接收一个底数参数，即以什么底数来输出数值的字符串表示。默认情况下，toString()返回数值的十进制字符串表示。而通过传入参数，可以得到数值的二进制、八进制、十六进制，或者其他任何有效基数的字符串表示，比如：

```
let num = 10;
console.log(num.toString());      // "10"
console.log(num.toString(2));     // "1010"
console.log(num.toString(8));     // "12"
console.log(num.toString(10));    // "10"
console.log(num.toString(16));    // "a"
```

这个例子展示了传入底数参数时，toString()输出的字符串值也会随之改变。数值 10 可以输出为任意数值格式。注意，默认情况下（不传参数）的输出与传入参数 10 得到的结果相同。

如果你不确定一个值是不是 null 或 undefined，可以使用 String()转型函数，它始终会返回表示相应类型值的字符串。String()函数遵循如下规则。

- 如果值有 toString()方法，则调用该方法（不传参数）并返回结果。
- 如果值是 null，返回"null"。
- 如果值是 undefined，返回"undefined"。

下面看几个例子：

```
let value1 = 10;
let value2 = true;
let value3 = null;
let value4;

console.log(String(value1));  // "10"
console.log(String(value2));  // "true"
console.log(String(value3));  // "null"
console.log(String(value4));  // "undefined"
```

这里展示了将 4 个值转换为字符串的情况：一个数值、一个布尔值、一个 null 和一个 undefined。数值和布尔值的转换结果与调用 toString()相同。因为 null 和 undefined 没有 toString()方法，所以 String()方法就直接返回了这两个值的字面量文本。

> **注意**　用加号操作符给一个值加上一个空字符串""也可以将其转换为字符串（加号操作符本章后面会介绍）。

4. 模板字面量

ECMAScript 6 新增了使用模板字面量定义字符串的能力。与使用单引号或双引号不同，模板字面量保留换行字符，可以跨行定义字符串：

```
let myMultiLineString = 'first line\nsecond line';
let myMultiLineTemplateLiteral = `first line
second line`;

console.log(myMultiLineString);
// first line
// second line"

console.log(myMultiLineTemplateLiteral);
// first line
```

```
// second line

console.log(myMultiLineString === myMultiLinetemplateLiteral); // true
```

顾名思义，模板字面量在定义模板时特别有用，比如下面这个 HTML 模板：

```
let pageHTML = `
<div>
  <a href="#">
    <span>Jake</span>
  </a>
</div>`;
```


由于模板字面量会保持反引号内部的空格，因此在使用时要格外注意。格式正确的模板字符串看起来可能会缩进不当：

```
// 这个模板字面量在换行符之后有 25 个空格符
let myTemplateLiteral = `first line
                         second line`;
console.log(myTemplateLiteral.length);  // 47

// 这个模板字面量以一个换行符开头
let secondTemplateLiteral = `
first line
second line`;
console.log(secondTemplateLiteral[0] === '\n'); // true

// 这个模板字面量没有意料之外的字符
let thirdTemplateLiteral = `first line
second line`;
console.log(thirdTemplateLiteral);
// first line
// second line
```

5. 字符串插值

模板字面量最常用的一个特性是支持字符串插值，也就是可以在一个连续定义中插入一个或多个值。技术上讲，模板字面量不是字符串，而是一种特殊的 JavaScript 句法表达式，只不过求值后得到的是字符串。模板字面量在定义时立即求值并转换为字符串实例，任何插入的变量也会从它们最接近的作用域中取值。

字符串插值通过在`${}`中使用一个 JavaScript 表达式实现：

```
let value = 5;
let exponent = 'second';
// 以前，字符串插值是这样实现的：
let interpolatedString =
  value + ' to the ' + exponent + ' power is ' + (value * value);

// 现在，可以用模板字面量这样实现：
let interpolatedTemplateLiteral =
  `${ value } to the ${ exponent } power is ${ value * value }`;

console.log(interpolatedString);           // 5 to the second power is 25
console.log(interpolatedTemplateLiteral);  // 5 to the second power is 25
```

所有插入的值都会使用 `toString()`强制转型为字符串，而且任何 JavaScript 表达式都可以用于插值。嵌套的模板字符串无须转义：

```
console.log(`Hello, ${ `World` }!`);  // Hello, World!
```

将表达式转换为字符串时会调用 toString()：

```
let foo = { toString: () => 'World' };
console.log(`Hello, ${ foo }!`);      // Hello, World!
```

在插值表达式中可以调用函数和方法：

```
function capitalize(word) {
  return `${ word[0].toUpperCase() }${ word.slice(1) }`;
}
console.log(`${ capitalize('hello') }, ${ capitalize('world') }!`); // Hello, World!
```

此外，模板也可以插入自己之前的值：

```
let value = '';
function append() {
  value = `${value}abc`
  console.log(value);
}
append();  // abc
append();  // abcabc
append();  // abcabcabc
```

6. 模板字面量标签函数

模板字面量也支持定义**标签函数**（tag function），而通过标签函数可以自定义插值行为。标签函数会接收被插值记号分隔后的模板和对每个表达式求值的结果。

标签函数本身是一个常规函数，通过前缀到模板字面量来应用自定义行为，如下例所示。标签函数接收到的参数依次是原始字符串数组和对每个表达式求值的结果。这个函数的返回值是对模板字面量求值得到的字符串。

最好通过一个例子来理解：

```
let a = 6;
let b = 9;

function simpleTag(strings, aValExpression, bValExpression, sumExpression) {
  console.log(strings);
  console.log(aValExpression);
  console.log(bValExpression);
  console.log(sumExpression);

  return 'foobar';
}

let untaggedResult = `${ a } + ${ b } = ${ a + b }`;
let taggedResult = simpleTag`${ a } + ${ b } = ${ a + b }`;
// ["", " + ", " = ", ""]
// 6
// 9
// 15

console.log(untaggedResult);   // "6 + 9 = 15"
console.log(taggedResult);     // "foobar"
```

因为表达式参数的数量是可变的，所以通常应该使用剩余操作符（rest operator）将它们收集到一个数组中：

```
let a = 6;
let b = 9;

function simpleTag(strings, ...expressions) {
  console.log(strings);
  for(const expression of expressions) {
    console.log(expression);
  }

  return 'foobar';
}
let taggedResult = simpleTag`${ a } + ${ b } = ${ a + b }`;
// ["", " + ", " = ", ""]
// 6
// 9
// 15

console.log(taggedResult);  // "foobar"
```

对于有 *n* 个插值的模板字面量，传给标签函数的表达式参数的个数始终是 *n*，而传给标签函数的第一个参数所包含的字符串个数则始终是 *n*+1。因此，如果你想把这些字符串和对表达式求值的结果拼接起来作为默认返回的字符串，可以这样做：

```
let a = 6;
let b = 9;

function zipTag(strings, ...expressions) {
  return strings[0] +
         expressions.map((e, i) => `${e}${strings[i + 1]}`)
                    .join('');
}

let untaggedResult =    `${ a } + ${ b } = ${ a + b }`;
let taggedResult = zipTag`${ a } + ${ b } = ${ a + b }`;

console.log(untaggedResult);  // "6 + 9 = 15"
console.log(taggedResult);    // "6 + 9 = 15"
```

7. 原始字符串

使用模板字面量也可以直接获取原始的模板字面量内容（如换行符或 Unicode 字符），而不是被转换后的字符表示。为此，可以使用默认的 `String.raw` 标签函数：

```
// Unicode 示例
// \u00A9 是版权符号
console.log(`\u00A9`);             // ©
console.log(String.raw`\u00A9`);  // \u00A9

// 换行符示例
console.log(`first line\nsecond line`);
// first line
// second line

console.log(String.raw`first line\nsecond line`); // "first line\nsecond line"

// 对实际的换行符来说是不行的
// 它们不会被转换成转义序列的形式
console.log(`first line
```

```
second line`);
// first line
// second line

console.log(String.raw`first line
second line`);
// first line
// second line
```

另外，也可以通过标签函数的第一个参数，即字符串数组的.raw 属性取得每个字符串的原始内容：

```
function printRaw(strings) {
  console.log('Actual characters:');
  for (const string of strings) {
    console.log(string);
  }

  console.log('Escaped characters;');
  for (const rawString of strings.raw) {
    console.log(rawString);
  }
}

printRaw`\u00A9${ 'and' }\n`;
// Actual characters:
// ©
// (换行符)
// Escaped characters:
// \u00A9
// \n
```

3.4.7　Symbol 类型

Symbol（符号）是 ECMAScript 6 新增的数据类型。符号是原始值，且符号实例是唯一、不可变的。符号的用途是确保对象属性使用唯一标识符，不会发生属性冲突的危险。

尽管听起来跟私有属性有点类似，但符号并不是为了提供私有属性的行为才增加的（尤其是因为 Object API 提供了方法，可以更方便地发现符号属性）。相反，符号就是用来创建唯一记号，进而用作非字符串形式的对象属性。

1. 符号的基本用法

符号需要使用 Symbol()函数初始化。因为符号本身是原始类型，所以 typeof 操作符对符号返回 symbol。

```
let sym = Symbol();
console.log(typeof sym); // symbol
```

调用 Symbol()函数时，也可以传入一个字符串参数作为对符号的描述（description），将来可以通过这个字符串来调试代码。但是，这个字符串参数与符号定义或标识完全无关：

```
let genericSymbol = Symbol();
let otherGenericSymbol = Symbol();

let fooSymbol = Symbol('foo');
let otherFooSymbol = Symbol('foo');

console.log(genericSymbol == otherGenericSymbol);  // false
```

```
console.log(fooSymbol == otherFooSymbol);          // false
```

符号没有字面量语法，这也是它们发挥作用的关键。按照规范，你只要创建 Symbol()实例并将其用作对象的新属性，就可以保证它不会覆盖已有的对象属性，无论是符号属性还是字符串属性。

```
let genericSymbol = Symbol();
console.log(genericSymbol);  // Symbol()

let fooSymbol = Symbol('foo');
console.log(fooSymbol);      // Symbol(foo);
```


最重要的是，Symbol()函数不能与 new 关键字一起作为构造函数使用。这样做是为了避免创建符号包装对象，像使用 Boolean、String 或 Number 那样，它们都支持构造函数且可用于初始化包含原始值的包装对象：

```
let myBoolean = new Boolean();
console.log(typeof myBoolean); // "object"

let myString = new String();
console.log(typeof myString);  // "object"

let myNumber = new Number();
console.log(typeof myNumber);  // "object"

let mySymbol = new Symbol(); // TypeError: Symbol is not a constructor
```

如果你确实想使用符号包装对象，可以借用 Object()函数：

```
let mySymbol = Symbol();
let myWrappedSymbol = Object(mySymbol);
console.log(typeof myWrappedSymbol);   // "object"
```

2. 使用全局符号注册表

如果运行时的不同部分需要共享和重用符号实例，那么可以用一个字符串作为键，在全局符号注册表中创建并重用符号。

为此，需要使用 Symbol.for()方法：

```
let fooGlobalSymbol = Symbol.for('foo');
console.log(typeof fooGlobalSymbol); // symbol
```

Symbol.for()对每个字符串键都执行幂等操作。第一次使用某个字符串调用时，它会检查全局运行时注册表，发现不存在对应的符号，于是就会生成一个新符号实例并添加到注册表中。后续使用相同字符串的调用同样会检查注册表，发现存在与该字符串对应的符号，然后就会返回该符号实例。

```
let fooGlobalSymbol = Symbol.for('foo');        // 创建新符号
let otherFooGlobalSymbol = Symbol.for('foo');  // 重用已有符号

console.log(fooGlobalSymbol === otherFooGlobalSymbol);  // true
```

即使采用相同的符号描述，在全局注册表中定义的符号跟使用 Symbol()定义的符号也并不等同：

```
let localSymbol = Symbol('foo');
let globalSymbol = Symbol.for('foo');

console.log(localSymbol === globalSymbol); // false
```

全局注册表中的符号必须使用字符串键来创建，因此作为参数传给 Symbol.for()的任何值都会被