与逻辑与类似，如果有一个操作数不是布尔值，那么逻辑或操作符也不一定返回布尔值。它遵循如下规则。

- 如果第一个操作数是对象，则返回第一个操作数。
- 如果第一个操作数求值为 false，则返回第二个操作数。
- 如果两个操作数都是对象，则返回第一个操作数。
- 如果两个操作数都是 null，则返回 null。
- 如果两个操作数都是 NaN，则返回 NaN。
- 如果两个操作数都是 undefined，则返回 undefined。

同样与逻辑与类似，逻辑或操作符也具有短路的特性。只不过对逻辑或而言，第一个操作数求值为 true，第二个操作数就不会再被求值了。看下面的例子：

```
let found = true;
let result = (found || someUndeclaredVariable); // 不会出错
console.log(result); // 会执行
```

跟前面的例子一样，变量 someUndeclaredVariable 也没有定义。但是，因为变量 found 的值为 true，所以逻辑或操作符不会对变量 someUndeclaredVariable 求值，而直接返回 true。假如把 found 的值改为 false，那就会报错了：

```
let found = false;
let result = (found || someUndeclaredVariable); // 这里会出错
console.log(result); // 不会执行这一行
```

利用这个行为，可以避免给变量赋值 null 或 undefined。比如：

```
let myObject = preferredObject || backupObject;
```

在这个例子中，变量 myObject 会被赋予两个值中的一个。其中，preferredObject 变量包含首选的值，backupObject 变量包含备用的值。如果 preferredObject 不是 null，则它的值就会赋给 myObject；如果 preferredObject 是 null，则 backupObject 的值就会赋给 myObject。这种模式在 ECMAScript 代码中经常用于变量赋值，本书后面的代码示例中也会经常用到。

3.5.4　乘性操作符

ECMAScript 定义了 3 个乘性操作符：乘法、除法和取模。这些操作符跟它们在 Java、C 语言及 Perl 中对应的操作符作用一样，但在处理非数值时，它们也会包含一些自动的类型转换。如果乘性操作符有不是数值的操作数，则该操作数会在后台被使用 Number()转型函数转换为数值。这意味着空字符串会被当成 0，而布尔值 true 会被当成 1。

1. 乘法操作符

乘法操作符由一个星号（*）表示，可以用于计算两个数值的乘积。其语法类似于 C 语言，比如：

```
let result = 34 * 56;
```

不过，乘法操作符在处理特殊值时也有一些特殊的行为。

- 如果操作数都是数值，则执行常规的乘法运算，即两个正值相乘是正值，两个负值相乘也是正值，正负符号不同的值相乘得到负值。如果 ECMAScript 不能表示乘积，则返回 Infinity 或 -Infinity。
- 如果有任一操作数是 NaN，则返回 NaN。

- 如果是 Infinity 乘以 0，则返回 NaN。
- 如果是 Infinity 乘以非 0 的有限数值，则根据第二个操作数的符号返回 Infinity 或-Infinity。
- 如果是 Infinity 乘以 Infinity，则返回 Infinity。
- 如果有不是数值的操作数，则先在后台用 Number()将其转换为数值，然后再应用上述规则。

2. 除法操作符

除法操作符由一个斜杠（/）表示，用于计算第一个操作数除以第二个操作数的商，比如：

```
let result = 66 / 11;
```

跟乘法操作符一样，除法操作符针对特殊值也有一些特殊的行为。

- 如果操作数都是数值，则执行常规的除法运算，即两个正值相除是正值，两个负值相除也是正值，符号不同的值相除得到负值。如果ECMAScript不能表示商，则返回 Infinity 或-Infinity。
- 如果有任一操作数是 NaN，则返回 NaN。
- 如果是 Infinity 除以 Infinity，则返回 NaN。
- 如果是 0 除以 0，则返回 NaN。
- 如果是非 0 的有限值除以 0，则根据第一个操作数的符号返回 Infinity 或-Infinity。
- 如果是 Infinity 除以任何数值，则根据第二个操作数的符号返回 Infinity 或-Infinity。
- 如果有不是数值的操作数，则先在后台用 Number()函数将其转换为数值，然后再应用上述规则。

3. 取模操作符

取模（余数）操作符由一个百分比符号（%）表示，比如：

```
let result = 26 % 5; // 等于 1
```

与其他乘性操作符一样，取模操作符对特殊值也有一些特殊的行为。

- 如果操作数是数值，则执行常规除法运算，返回余数。
- 如果被除数是无限值，除数是有限值，则返回 NaN。
- 如果被除数是有限值，除数是 0，则返回 NaN。
- 如果是 Infinity 除以 Infinity，则返回 NaN。
- 如果被除数是有限值，除数是无限值，则返回被除数。
- 如果被除数是 0，除数不是 0，则返回 0。
- 如果有不是数值的操作数，则先在后台用 Number()函数将其转换为数值，然后再应用上述规则。

3.5.5 指数操作符

ECMAScript 7 新增了指数操作符，Math.pow()现在有了自己的操作符**，结果是一样的：

```
console.log(Math.pow(3, 2);     // 9
console.log(3 ** 2);            // 9

console.log(Math.pow(16, 0.5); // 4
console.log(16** 0.5);          // 4
```

不仅如此，指数操作符也有自己的指数赋值操作符**=，该操作符执行指数运算和结果的赋值操作：

```
let squared = 3;
squared **= 2;
console.log(squared); // 9
```

```
let sqrt = 16;
sqrt **= 0.5;
console.log(sqrt); // 4
```

3.5.6　加性操作符

加性操作符，即加法和减法操作符，一般都是编程语言中最简单的操作符。不过，在 ECMAScript 中，这两个操作符拥有一些特殊的行为。与乘性操作符类似，加性操作符在后台会发生不同数据类型的转换。只不过对这两个操作符来说，转换规则不是那么直观。

1. 加法操作符

加法操作符（+）用于求两个数的和，比如：

```
let result = 1 + 2;
```

如果两个操作数都是数值，加法操作符执行加法运算并根据如下规则返回结果：

- 如果有任一操作数是 NaN，则返回 NaN；
- 如果是 Infinity 加 Infinity，则返回 Infinity；
- 如果是-Infinity 加-Infinity，则返回-Infinity；
- 如果是 Infinity 加-Infinity，则返回 NaN；
- 如果是+0 加+0，则返回+0；
- 如果是-0 加+0，则返回+0；
- 如果是-0 加-0，则返回-0。

不过，如果有一个操作数是字符串，则要应用如下规则：

- 如果两个操作数都是字符串，则将第二个字符串拼接到第一个字符串后面；
- 如果只有一个操作数是字符串，则将另一个操作数转换为字符串，再将两个字符串拼接在一起。

如果有任一操作数是对象、数值或布尔值，则调用它们的 toString()方法以获取字符串，然后再应用前面的关于字符串的规则。对于 undefined 和 null，则调用 String()函数，分别获取"undefined"和"null"。

看下面的例子：

```
let result1 = 5 + 5;        // 两个数值
console.log(result1);       // 10
let result2 = 5 + "5";      // 一个数值和一个字符串
console.log(result2);       // "55"
```

以上代码展示了加法操作符的两种运算模式。正常情况下，5 + 5 等于 10（数值），如前两行代码所示。但是，如果将一个操作数改为字符串，比如"5"，则相加的结果就变成了"55"（原始字符串值），因为第一个操作数也会被转换为字符串。

ECMAScript 中最常犯的一个错误，就是忽略加法操作中涉及的数据类型。比如下面这个例子：

```
let num1 = 5;
let num2 = 10;
let message = "The sum of 5 and 10 is " + num1 + num2;
console.log(message);  // "The sum of 5 and 10 is 510"
```

这里，变量 message 中保存的是一个字符串，是执行两次加法操作之后的结果。有人可能会认为最终得到的字符串是"The sum of 5 and 10 is 15"。可是，实际上得到的是"The sum of 5 and 10 is 510"。这是因为每次加法运算都是独立完成的。第一次加法的操作数是一个字符串和一个数值（5），

结果还是一个字符串。第二次加法仍然是用一个字符串去加一个数值（10），同样也会得到一个字符串。如果想真正执行数学计算，然后把结果追加到字符串末尾，只要使用一对括号即可：

```
let num1 = 5;
let num2 = 10;
let message = "The sum of 5 and 10 is " + (num1 + num2);
console.log(message); // "The sum of 5 and 10 is 15"
```

在此，我们用括号把两个数值变量括了起来，意思是让解释器先执行两个数值的加法，然后再把结果追加给字符串。因此，最终得到的字符串变成了"The sum of 5 and 10 is 15"。

2. 减法操作符

减法操作符（-）也是使用很频繁的一种操作符，比如：

```
let result = 2 - 1;
```

与加法操作符一样，减法操作符也有一组规则用于处理 ECMAScript 中不同类型之间的转换。

- 如果两个操作数都是数值，则执行数学减法运算并返回结果。
- 如果有任一操作数是 NaN，则返回 NaN。
- 如果是 Infinity 减 Infinity，则返回 NaN。
- 如果是-Infinity 减-Infinity，则返回 NaN。
- 如果是 Infinity 减-Infinity，则返回 Infinity。
- 如果是-Infinity 减 Infinity，则返回-Infinity。
- 如果是+0 减+0，则返回+0。
- 如果是+0 减-0，则返回-0。
- 如果是-0 减-0，则返回+0。
- 如果有任一操作数是字符串、布尔值、null 或 undefined，则先在后台使用 Number()将其转换为数值，然后再根据前面的规则执行数学运算。如果转换结果是 NaN，则减法计算的结果是 NaN。
- 如果有任一操作数是对象，则调用其 valueOf()方法取得表示它的数值。如果该值是 NaN，则减法计算的结果是 NaN。如果对象没有 valueOf()方法，则调用其 toString()方法，然后再将得到的字符串转换为数值。

以下示例演示了上面的规则：

```
let result1 = 5 - true; // true 被转换为 1，所以结果是 4
let result2 = NaN - 1;  // NaN
let result3 = 5 - 3;    // 2
let result4 = 5 - "";   // ""被转换为 0，所以结果是 5
let result5 = 5 - "2";  // "2"被转换为 2，所以结果是 3
let result6 = 5 - null; // null 被转换为 0，所以结果是 5
```

3.5.7 关系操作符

关系操作符执行比较两个值的操作，包括小于（<）、大于（>）、小于等于（<=）和大于等于（>=），用法跟数学课上学的一样。这几个操作符都返回布尔值，如下所示：

```
let result1 = 5 > 3; // true
let result2 = 5 < 3; // false
```

与 ECMAScript 中的其他操作符一样，在将它们应用到不同数据类型时也会发生类型转换和其他行为。

- 如果操作数都是数值，则执行数值比较。
- 如果操作数都是字符串，则逐个比较字符串中对应字符的编码。
- 如果有任一操作数是数值，则将另一个操作数转换为数值，执行数值比较。
- 如果有任一操作数是对象，则调用其 `valueOf()`方法，取得结果后再根据前面的规则执行比较。如果没有 `valueOf()`操作符，则调用 `toString()`方法，取得结果后再根据前面的规则执行比较。
- 如果有任一操作数是布尔值，则将其转换为数值再执行比较。

在使用关系操作符比较两个字符串时，会发生一个有趣的现象。很多人认为小于意味着“字母顺序靠前”，而大于意味着“字母顺序靠后”，实际上不是这么回事。对字符串而言，关系操作符会比较字符串中对应字符的编码，而这些编码是数值。比较完之后，会返回布尔值。问题的关键在于，大写字母的编码都小于小写字母的编码，因此以下这种情况就会发生：

```
let result = "Brick" < "alphabet"; // true
```

在这里，字符串`"Brick"`被认为小于字符串`"alphabet"`，因为字母 B 的编码是 66，字母 a 的编码是 97。要得到确实按字母顺序比较的结果，就必须把两者都转换为相同的大小写形式（全大写或全小写），然后再比较：

```
let result = "Brick".toLowerCase() < "alphabet".toLowerCase(); // false
```

将两个操作数都转换为小写，就能保证按照字母表顺序判定`"alphabet"`在`"Brick"`前头。

另一个奇怪的现象是在比较两个数值字符串的时候，比如下面这个例子：

```
let result = "23" < "3"; // true
```

这里在比较字符串`"23"`和`"3"`时返回 `true`。因为两个操作数都是字符串，所以会逐个比较它们的字符编码（字符`"2"`的编码是 50，而字符`"3"`的编码是 51）。不过，如果有一个操作数是数值，那么比较的结果就对了：

```
let result = "23" < 3; // false
```

因为这次会将字符串`"23"`转换为数值 23，然后再跟 3 比较，结果当然对了。只要是数值和字符串比较，字符串就会先被转换为数值，然后进行数值比较。对于数值字符串而言，这样能保证结果正确。但如果字符串不能转换成数值呢？比如下面这个例子：

```
let result = "a" < 3; // 因为"a"会转换为 NaN，所以结果是 false
```

因为字符`"a"`不能转换成任何有意义的数值，所以只能转换为 `NaN`。这里有一个规则，即任何关系操作符在涉及比较 `NaN` 时都返回 `false`。这样一来，下面的例子有趣了：

```
let result1 = NaN < 3;  // false
let result2 = NaN >= 3; // false
```

在大多数比较的场景中，如果一个值不小于另一个值，那就一定大于或等于它。但在比较 `NaN` 时，无论是小于还是大于等于，比较的结果都会返回 `false`。

3.5.8　相等操作符

判断两个变量是否相等是编程中最重要的操作之一。在比较字符串、数值和布尔值是否相等时，过程都很直观。但是在比较两个对象是否相等时，情形就比较复杂了。ECMAScript 中的相等和不相等操作符，原本在比较之前会执行类型转换，但很快就有人质疑这种转换是否应该发生。最终，ECMAScript

提供了两组操作符。第一组是**等于**和**不等于**，它们在比较之前执行转换。第二组是**全等**和**不全等**，它们在比较之前不执行转换。

1. 等于和不等于

ECMAScript 中的等于操作符用两个等于号（==）表示，如果操作数相等，则会返回 true。不等于操作符用叹号和等于号（!=）表示，如果两个操作数不相等，则会返回 true。这两个操作符都会先进行类型转换（通常称为**强制类型转换**）再确定操作数是否相等。

在转换操作数的类型时，相等和不相等操作符遵循如下规则。

- 如果任一操作数是布尔值，则将其转换为数值再比较是否相等。false 转换为 0，true 转换为 1。
- 如果一个操作数是字符串，另一个操作数是数值，则尝试将字符串转换为数值，再比较是否相等。
- 如果一个操作数是对象，另一个操作数不是，则调用对象的 valueOf()方法取得其原始值，再根据前面的规则进行比较。

在进行比较时，这两个操作符会遵循如下规则。

- null 和 undefined 相等。
- null 和 undefined 不能转换为其他类型的值再进行比较。
- 如果有任一操作数是 NaN，则相等操作符返回 false，不相等操作符返回 true。记住：即使两个操作数都是 NaN，相等操作符也返回 false，因为按照规则，NaN 不等于 NaN。
- 如果两个操作数都是对象，则比较它们是不是同一个对象。如果两个操作数都指向同一个对象，则相等操作符返回 true。否则，两者不相等。

下表总结了一些特殊情况及比较的结果。

表 达 式	结 果
null == undefined	true
"NaN" == NaN	false
5 == NaN	false
NaN == NaN	false
NaN != NaN	true
false == 0	true
true == 1	true
true == 2	false
undefined == 0	false
null == 0	false
"5" == 5	true

2. 全等和不全等

全等和不全等操作符与相等和不相等操作符类似，只不过它们在比较相等时不转换操作数。全等操作符由 3 个等于号（===）表示，只有两个操作数在不转换的前提下相等才返回 true，比如：

```
let result1 = ("55" == 55);   // true，转换后相等
let result2 = ("55" === 55);  // false，不相等，因为数据类型不同
```

在这个例子中，第一个比较使用相等操作符，比较的是字符串"55"和数值 55。如前所述，因为字

符串"55"会被转换为数值 55，然后再与数值 55 进行比较，所以返回 true。第二个比较使用全等操作符，因为没有转换，字符串和数值当然不能相等，所以返回 false。

不全等操作符用一个叹号和两个等于号（!==）表示，只有两个操作数在不转换的前提下不相等才返回 true。比如：

```
let result1 = ("55" != 55);  // false, 转换后相等
let result2 = ("55" !== 55); // true, 不相等, 因为数据类型不同
```

这一次，第一个比较使用不相等操作符，它会把字符串"55"转换为数值 55，跟第二个操作数相等。既然转换后两个值相等，那就返回 false。第二个比较使用不全等操作符。这时候可以这么问："字符串 55 和数值 55 有区别吗？"答案是"有"（true）。

另外，虽然 null == undefined 是 true（因为这两个值类似），但 null === undefined 是 false，因为它们不是相同的数据类型。

> **注意**　由于相等和不相等操作符存在类型转换问题，因此推荐使用全等和不全等操作符。这样有助于在代码中保持数据类型的完整性。

3.5.9　条件操作符

条件操作符是 ECMAScript 中用途最为广泛的操作符之一，语法跟 Java 中一样：

```
variable = boolean_expression ? true_value : false_value;
```

上面的代码执行了条件赋值操作，即根据条件表达式 boolean_expression 的值决定将哪个值赋给变量 variable。如果 boolean_expression 是 true，则赋值 true_value；如果 boolean_expression 是 false，则赋值 false_value。比如：

```
let max = (num1 > num2) ? num1 : num2;
```

在这个例子中，max 将被赋予一个最大值。这个表达式的意思是，如果 num1 大于 num2（条件表达式为 true），则将 num1 赋给 max。否则，将 num2 赋给 max。

3.5.10　赋值操作符

简单赋值用等于号（=）表示，将右手边的值赋给左手边的变量，如下所示：

```
let num = 10;
```

复合赋值使用乘性、加性或位操作符后跟等于号（=）表示。这些赋值操作符是类似如下常见赋值操作的简写形式：

```
let num = 10;
num = num + 10;
```

以上代码的第二行可以通过复合赋值来完成：

```
let num = 10;
num += 10;
```

每个数学操作符以及其他一些操作符都有对应的复合赋值操作符：

- 乘后赋值（*=）
- 除后赋值（/=）
- 取模后赋值（%=）
- 加后赋值（+=）
- 减后赋值（-=）
- 左移后赋值（<<=）
- 右移后赋值（>>=）
- 无符号右移后赋值（>>>=）

这些操作符仅仅是简写语法，使用它们不会提升性能。

3.5.11 逗号操作符

逗号操作符可以用来在一条语句中执行多个操作，如下所示：

```
let num1 = 1, num2 = 2, num3 = 3;
```

在一条语句中同时声明多个变量是逗号操作符最常用的场景。不过，也可以使用逗号操作符来辅助赋值。在赋值时使用逗号操作符分隔值，最终会返回表达式中最后一个值：

```
let num = (5, 1, 4, 8, 0); // num 的值为 0
```

在这个例子中，num 将被赋值为 0，因为 0 是表达式中最后一项。逗号操作符的这种使用场景并不多见，但这种行为的确存在。

3.6 语句

ECMA-262 描述了一些语句（也称为**流控制语句**），而 ECMAScript 中的大部分语法都体现在语句中。语句通常使用一或多个关键字完成既定的任务。语句可以简单，也可以复杂。简单的如告诉函数退出，复杂的如列出一堆要重复执行的指令。

3.6.1 if 语句

if 语句是使用最频繁的语句之一，语法如下：

```
if (condition) statement1 else statement2
```

这里的条件（condition）可以是任何表达式，并且求值结果不一定是布尔值。ECMAScript 会自动调用 Boolean() 函数将这个表达式的值转换为布尔值。如果条件求值为 true，则执行语句 statement1；如果条件求值为 false，则执行语句 statement2。这里的语句可能是一行代码，也可能是一个代码块（即包含在一对花括号中的多行代码）。来看下面的例子：

```
if (i > 25)
  console.log("Greater than 25."); // 只有一行代码的语句
else {
  console.log("Less than or equal to 25."); // 一个语句块
}
```

这里的最佳实践是使用语句块，即使只有一行代码要执行也是如此。这是因为语句块可以避免对什么条件下执行什么产生困惑。

可以像这样连续使用多个 if 语句：

```
if (condition1) statement1 else if (condition2) statement2 else statement3
```

下面是一个例子：

```
if (i > 25) {
  console.log("Greater than 25.");
} else if (i < 0) {
  console.log("Less than 0.");
} else {
  console.log("Between 0 and 25, inclusive.");
}
```

3.6.2　do-while 语句

do-while 语句是一种后测试循环语句，即循环体中的代码执行后才会对退出条件进行求值。换句话说，循环体内的代码至少执行一次。do-while 的语法如下：

```
do {
  statement
} while (expression);
```

下面是一个例子：

```
let i = 0;
do {
  i += 2;
} while (i < 10);
```

在这个例子中，只要 i 小于 10，循环就会重复执行。i 从 0 开始，每次循环递增 2。

> **注意**　后测试循环经常用于这种情形：循环体内代码在退出前至少要执行一次。

3.6.3　while 语句

while 语句是一种先测试循环语句，即先检测退出条件，再执行循环体内的代码。因此，while 循环体内的代码有可能不会执行。下面是 while 循环的语法：

```
while(expression) statement
```

这是一个例子：

```
let i = 0;
while (i < 10) {
  i += 2;
}
```

在这个例子中，变量 i 从 0 开始，每次循环递增 2。只要 i 小于 10，循环就会继续。

3.6.4　for 语句

for 语句也是先测试语句，只不过增加了进入循环之前的初始化代码，以及循环执行后要执行的表达式，语法如下：

```
for (initialization; expression; post-loop-expression) statement
```

下面是一个用例：

```
let count = 10;
for (let i = 0; i < count; i++) {
  console.log(i);
}
```

以上代码在循环开始前定义了变量 i 的初始值为 0。然后求值条件表达式，如果求值结果为 true（i < count），则执行循环体。因此循环体也可能不会被执行。如果循环体被执行了，则循环后表达式也会执行，以便递增变量 i。for 循环跟下面的 while 循环是一样的：

```
let count = 10;
let i = 0;
while (i < count) {
  console.log(i);
  i++;
}
```

无法通过 while 循环实现的逻辑，同样也无法使用 for 循环实现。因此 for 循环只是将循环相关的代码封装在了一起而已。

在 for 循环的初始化代码中，其实是可以不使用变量声明关键字的。不过，初始化定义的迭代器变量在循环执行完成后几乎不可能再用到了。因此，最清晰的写法是使用 let 声明迭代器变量，这样就可以将这个变量的作用域限定在循环中。

初始化、条件表达式和循环后表达式都不是必需的。因此，下面这种写法可以创建一个无穷循环：

```
for (;;) { // 无穷循环
  doSomething();
}
```

如果只包含条件表达式，那么 for 循环实际上就变成了 while 循环：

```
let count = 10;
let i = 0;
for (; i < count; ) {
  console.log(i);
  i++;
}
```

这种多功能性使得 for 语句在这门语言中使用非常广泛。

3.6.5 for-in 语句

for-in 语句是一种严格的迭代语句，用于枚举对象中的非符号键属性，语法如下：

```
for (property in expression) statement
```

下面是一个例子：

```
for (const propName in window) {
  document.write(propName);
}
```

这个例子使用 for-in 循环显示了 BOM 对象 window 的所有属性。每次执行循环，都会给变量 propName 赋予一个 window 对象的属性作为值，直到 window 的所有属性都被枚举一遍。与 for 循环一样，这里控制语句中的 const 也不是必需的。但为了确保这个局部变量不被修改，推荐使用 const。

ECMAScript 中对象的属性是无序的，因此 for-in 语句不能保证返回对象属性的顺序。换句话说，