## 逻辑

### 布尔值

小的时候，我们认为一张白纸只有两面，一个人只有好人坏人的区别，这个世界只有黑白两种颜色，说的话只有真假之分。现在，我们也可以说：逻辑——所有的逻辑，本质上都是一个靠真与假驱动的世界。真，则一往直前；假，则回归起点。JavaScript 提供了 true 来表示所有的“真”，false 来表示所有的“假”。他们便是一个真与假的二元世界。true 和 false 被称为布尔值，用以纪念 19 世纪为逻辑学做出杰出贡献的 George Boole。

在 JavaScript 中，布尔值属于 boolean 类型。你可以直接使用它们来明确地表示真假、是非，也时常会隐藏在一串逻辑表达式中，作为它背后的力量。

alert(true); // true  
alert(false); // false

### undefined 和 null

先来看四个句子：

杨雨露有个姐姐，她叫杨雨晴。

杨雨露没有姐姐。

杨雨露不知道自己有没有姐姐。

杨雨露有姐姐，但是不知道姐姐在哪。

JavaScript 为第二种情况提供了 null，为第三、四种情况提供了 undefined。

null 和 undefined 是 JavaScript 定义的两个特殊值，分别表示

1. 一个空值。

* 这个可能需要一个值，但是明确地知道”这是空的“，用 null 来表示空值。

1. 未发现需要的值。

* 这个地方不知道有没有值，用 undefined 来表示”未定义“。

null 是 JavaScript 的*关键字*，如果对它进行声明或赋值操作会产生错误。

let null; // SyntaxError: Unexpected token null  
null = 1; // ReferenceError: Invalid left-hand side in assignment

undefined 不是一个明确定义的保留字，如果尝试对它赋值不会产生错误，但它的值也不会改变。

alert(undefined); // undefined  
undefined = 0; // 不会产生错误  
alert(undefined); // undefined

如果对它进行重复声明，则它会被视作一个变量，在声明前会产生一个“变量未定义”的错误，在声明后使用它则会发现它的值已经发生改变。这是一个语言缺陷。

alert(undefined); // ReferenceError: undefined is not defined  
let undefined = 0;  
alert(undefined)； // 0  
undefined = 3;  
alert(undefined); // 3

我们可以使用表达式 void 0 来得到最“纯粹”的 undefined 值。并且我们也推荐这种方法——它写起来更简短！

alert(void 0); // undefined  
undefined = 0;  
alert(undefined); // 0  
alert(void 0); // undefined

如果我们声明了一个变量却没有给它赋予任何值，那么它的默认值就是 undefined ——即“未定义”。

let a;  
alert(a); // undefined

练习 3.1.1

1. 举出生活中可以分别用 null 和 undefined 描述的例子。
2. 尝试了解在 JavaScript 的创造过程中，null 和 undefined 分别是怎样出现的。

### 逻辑运算

很多人觉得逻辑冰冷而机械死板，正是如此。因此，它才有用。人类易被感情左右，但计算机不同。正是因为冰冷且机械死板，计算机才会一直稳定运行，为我们所用。

逻辑的本质是真与假的组合。在 JavaScript 中，以下值都会被视为“假”：

false NaN 0 "" '' null undefined

除了以上的“假”值，其他自然都是“真”值。关于 0 、特殊数值 NaN、字符串的概念将在下文中讲到。

真值都可以被看做 true，假值都可以被看做 false。这两个布尔值是逻辑的基本组成部分，简单的逻辑自然也可以组合成更复杂的逻辑，这个组合的过程我们称为*逻辑运算*。与、或、非是三个基本的逻辑运算，JavaScript 提供了它们的运算符 && || 和 ！。这三个运算得到的值与参与运算的值有关，但是得到的还是参与运算的值本身，而不一定是布尔值。

1. **&& （与）**

* 如果两个条件都为 true，那么得到 true，否则得到 false。

它和我们平时说话时“如果……并且……”是类似的，即判断两个条件是否都*成立*，

alert(true && false); // false  
alert(true && true); // true  
alert(false && false); // false

很容易理解，对吧！但事实上，&& 不一定会得到一个布尔值。它得到的值与用来运算的值有关，如果运算的值不是布尔值，它也不一定得到一个布尔值，而是根据值本身被看做“真”或被看做“假”来决定得到什么值。

它的具体运算方式如下：

* 如果第一个条件被视为 true，而第二个条件被视为 false ，那么得到第二个条件的值。
* 如果两个条件都被视为 true，那么得到第二个条件的值。
* 如果第一个条件被视为 false，那么得到第一个条件的值。

示例：

alert(0 && true); // 0  
alert(true && 0); // 0  
alert(0 && false); // 0  
alert(false && 0); // false  
alert(100 && 0); // 0  
alert("Hello" && "") // ""  
alert(null && undefined) // null  
alert(100 && NaN) // NaN

1. **||（或）**

* 两个条件中只要有一个为 true，那么得到 true，否则为 false。

它和我们平时所说的“如果……或者……”是等价的。

alert(true || false); // true（第一个条件的值）  
alert(true || true); // true（第一个条件的值）  
alert(false || false); // false（第二个条件的值）

和 && 类似，|| 也不一定得到一个布尔值，而是根据它所运算的值被看做“真”还是看做“假”来得到值。

它的具体运算方式如下：

* 如果第一个条件被视为 true，那么直接得到第一个条件的值。
* 如果第一个条件被视为false，那么得到第二个条件的值。

示例：

alert(0 || true); // true  
alert(true || 0); // true  
alert(0 || false); // false  
alert(false || 0); // 0  
alert(100 || 0); // 100  
alert("Hello" || "") // "Hello"  
alert(null || undefined) // undefined  
alert(100 || NaN) // 100

换句话说，如果第一个条件为“真”，那么就符合“或”的条件了，不必再判断下一个。如果第一个条件为假，就需要将第二个条件作为整个运算得到的值。

1. **!（非）**

* 如果值为 true，那么得到 false，否则得到 true。

它确实得到一个布尔值，具体运算方式如下：

* 如果条件被视为 true，那么得到 false。
* 如果条件被视为 false，那么得到 true。

示例：

alert(!true); // false  
alert(!false); // true  
alert(!0); // true  
alert(!100); // false  
alert(!NaN); // true  
alert(!""); // true  
alert(!undefined); // true  
alert(!!0); // false  
alert(!!null); // true

因此，!! 两个非运算重复进行，得到的值就是条件本身的布尔值描述形式，即被看做“真”还是“假”。

**运算符优先级**

当 && || ! 三个运算符同时在一个表达式中，运算过程遵循**操作符优先级**。! 操作符具有最高的优先级，即在一个表达式中它所属的式子总是被最先计算，其次是 &&，|| 的优先级最低。

alert(10 && !5); // false  
alert(!5 && 10); // false  
alert(!5 || 6 && 7); // 7  
alert(5 || 6 && 7 || 8); // 5  
alert(5 && 6 || 7 && 8); // 6  
alert(!5 || !6 || 7 && 8); // 8

在第三行代码中：

1. !5 由于具有最高的优先级，被最先计算。由于它是 false，且是 || 的第一个条件，因此会继续计算位于 || 右侧的第二个条件。
2. 由于 && 的优先级大于 || ，因此会计算 6 && 7，结果为 7，那么 || 的第二个条件就是 7。
3. 因此整个逻辑表达式的结果就是 7。

在第四行代码中：

1. 5 被视作 true, 因此 || 运算符不会查看第二个条件。
2. 结果为 5。

在第五行代码中，首先从左往右运算，5 && 6 的值为 6，则 || 的第一个条件为 6，最后结果为 6。

在第六行代码中：

1. !5 为 false，|| 运算符会查看第二个条件。
2. 由于 ! 具有最高的优先级，!6 会先得到计算，结果为 false。
3. 那么 !5 || !6 的值为 false。
4. 第二个 || 操作符会查看右边的条件。
5. 由于 && 的优先级大于 || ，会先计算 7 && 8 的值，值为 8。
6. 那么右边的条件为 8。
7. false || 8 的值为 8。

在实际应用中，我们可以使用括号 ( ) 来更改默认的运算符优先级。使用了括号的示例如下：

alert((5 || 6) && (7 || 8)); // 8  
alert(5 && (6 || 7) && 8); // 8  
alert(5 && 6 && (7 && 8)); // 8

一个操作符后使用括号括起来的内容是一个整体，会先计算括号中表达式的值，这个值作为该操作符的条件进行下一步计算。因此，在第一行代码中：

1. (5 || 6) 作为一个整体会被计算，值为 6。
2. 6 成为了 && 的第一个条件。&& 会查看第二个条件。
3. 第二个条件是 (7 || 8)，值为 8，所以第二个条件是 8。
4. 6 && 8 结果为 8。

你可以尝试演算另两行代码的运算过程。

练习 3.1.2

1. 计算如下代码的值：

* (18 || 24) && (15 && 0) || 6 || !12

1. 计算如下代码的值：

* !(15 || 0) && !(12 && !12)

1. 计算如下代码的值：

* 18 && (!(15 || 10) && (15 && 10))

### 条件表达式

条件表达式是一种*三目运算符*，它需要三个操作数。格式如下：

a ? b : c

如果 a 被视作 true，则这个表达式的值为 b ，否则为 c。

示例：

let a = 0;  
alert(a ? "Hello" : "Hi"); // "Hi"

条件表达式的运算符具有最低的优先级。也就是说，如果 a b c 都是其它表达式，那么一定会先计算出 a b c 的值，再得到条件表达式的值。一般来说，如果 a b c 都是表达式，我们推荐给用括号进行包裹以避免混淆。

练习 3.1.3

1. 计算下列条件表达式的值：

* (!(15 || 10) && (15 && 10)) ? "Hello" : "world"

1. 计算下列条件表达式的值：

* (15 || (true && NaN) || !Infinity && (!NaN || 12))) ? "Jim" : "Tom"