**CHAPTER 10 Booleans**

**第10章 Boolean类型**

The primitive boolean type comprises the values true and false:

基本的布尔类型包含2个值： true 和 false：

> typeof false

'boolean'

> typeof true

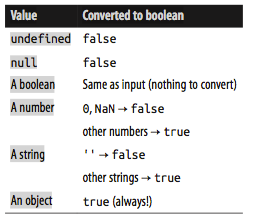
'boolean'

**Converting to Boolean**

**转换为布尔类型**

Values are converted to booleans as follows:

不同类型的值转换成的布尔值如下：



**Manually Converting to Boolean**

**手动转换为布尔类型**

There are three ways any value can be converted to a boolean:

有三种方式可以把任何值转换为布尔类型：

Boolean(value) (Invoked as a function, not as a constructor) 作为函数调用，而不是构造函数

 value ? true : false

 !!value A single “not” converts to negated boolean; use twice for the nonnegated conversion. 一次“非”转换为取反后的布尔值；两次“非”转换为对应的布尔值。

I prefer Boolean(), because it is more descriptive. Here are some examples:

我更喜欢 Boolean() 方式，因为它更具描述性。下面是一些例子：

> Boolean(undefined)

false

> Boolean(null)

false

> Boolean(0)

false

> Boolean(1)

true

> Boolean(2)

true

> Boolean('')

false

> Boolean('abc')

true

> Boolean('false')

true

**Truthy and Falsy Values**

**真（Truthy）值和假（Falsy）值**

Wherever JavaScript expects a boolean, you can provide any kind of value and it is automatically converted to boolean. Thus, there are two sets of values in JavaScript: one set is converted to false, while the other set is converted to true. These sets are called falsy values and truthy values. Given the preceding table, the following are all falsy values:

无论何时，JavaScript 执行过程中需要的是布尔值，它都可以将你提供的任何类型的值自动转换为布尔值。 因此，Javascript 有两类值：一类会转化为false，另一类会转化为ture。这两类被称为假（falsy）值和真（truthy）值。根据前面的表格，以下是所有的假值：

• undefined,null

• Boolean:false

• Number:0,NaN

• String:''

All other values—including all objects, even empty objects, empty arrays, and new Boolean(false)—are truthy. Because undefined and null are falsy, you can use the if statement to check whether a variable x has a value:

除去以上的值，所有其他值（包括所有的对象，甚至是空对象、空数组和new Boolean(false) )都是真。因为 undefined 和 null 都是假，可以用 if 语句来判断一个变量x是否有值：

if(x){

 // x has a value

}

The caveat is that the preceding check interprets all falsy values as “does not have a value,” not just undefined and null. But if you can live with that limitation, you get to use a compact and established pattern.

需要说明的是前面的这个判断把所有的假值解释为“没有值”，不仅仅是 undefined 和 null。但是，如果你可以接受这种限制，就可以使用这种简洁和既定的模式。

**Pitfall: all objects are truthy**

陷阱:所有的对象都是真

All objects are truthy:

所有的对象都是真

> Boolean(new Boolean(false))

true

> Boolean([])

true

> Boolean({})

true

That is different from how objects are converted to a number or string, where you can control the result by implementing the methods valueOf() and toString():

这不同于对象转换成数字或字符串的方式（可以通过实现 valueOf() 和 toString() 方法来控制结果）：

> Number({ valueOf: function () { return 123 } })

123

> String({ toString: function () { return 'abc' } })

'abc'

**History: Why are objects always truthy?**

历史:为什么对象总是真?

The conversion to boolean is different for historic reasons. For ECMAScript 1, it was decided to not enable objects to configure that conversion (e.g., via a toBoolean() method). The rationale was that the boolean operators || and && preserve the values of their operands. Therefore, if you chain those operators, the same value may be checked multiple times for truthiness or falsiness. Such checks are cheap for primitives, but would be costly for objects if they were able to configure their conversion to boolean. ECMAScript 1 avoided that cost by making objects always truthy.

转换为布尔类型与转换为其他类型的不同是由于历史原因造成的。ECMAScript 1决定不启用对象的这种转换配置 (例如，通过 toBoolean() 方法)。基本原理是布尔操作符 || 和 && 会保存其操作数的值。因此，如果你链式使用这些操作符，相同的值会被多次检查是真是假。这种检查对基本类型来说代价低廉，但对于可以配置转化为布尔值的对象来说代价高昂。所以ECMAScript 1通过使对象总是真来避免这种代价。

**Logical Operators**

**逻辑操作符**

In this section, we cover the basics of the And (&&), Or (||), and Not (!) logical operators.

在本节中，我们介绍了逻辑操作符 与(&&)、或（||）和 非(!) 的基础。

**Binary Logical Operators: And (&&) and Or (||)**

**二元逻辑操作符：与（&&） 和 或（||）**

Binary logical operators are:

二元逻辑操作符有如下特性：

Value-preserving

保留值

They always return either one of the operands, unchanged:

总是返回任何其中的一个操作数， 并不会改变这个操作数：

> 'abc' || 123

'abc'

> false || 123

123

Short-circuiting

短路

The second operand is not evaluated if the first operand already determines the result. For example (the result of console.log is undefined):

如果第一个操作数已经确定了结果，那么第二个操作数不会求值。例如（console.log 的结果是 undefined）

> true || console.log('Hello')

true

> false || console.log('Hello')

Hello

undefined

That is uncommon behavior for operators. Normally, all operands are evaluated before an operator is invoked (just like for functions).

这个行为对操作符来说不太寻常。通常，在操作符被调用前所有的操作数都会被求值（类似方法调用）。

**Logical And (&&)**

**逻辑与（&&）**

If the first operand can be converted to false, return it. Otherwise, return the second operand:

如果第一个操作数可以转换为 false，直接返回它。否则，返回第二个操作数:

> true && false

false

> false && 'def'

false

> '' && 'def'

''

> 'abc' && 'def'

'def'

**Logical Or (||)**

**逻辑或（||）**

If the first operand can be converted to true, return it. Otherwise, return the second operand:

如果第一个操作数可以转换为 true，直接返回它。否则，返回第二个操作数

> true || false

true

> true || 'def'

true

> 'abc' || 'def'

'abc'

> '' || 'def'

'def'

**Pattern: providing a default value**

**模式：提供一个默认值**

Sometimes there are situations where a value (a parameter, the result of a function, etc.) can be either a nonvalue (undefined, null) or an actual value. If you want to provide a default value for the former case, you can use the Or operator:

有时情况是，一个值（参数、方法的结果等）可以是空（nonvalue ：undefined， null）或一个真实的值。 如果你想为前一种情况提供一个默认值，可以使用 或（Or） 操作符。

theValue || defaultValue

The preceding expression evaluates to theValue if it is truthy and to defaultValue otherwise. The usual caveat applies: defaultValue will also be returned if theValue has a falsy value other than undefined and null. Let’s look at three examples of using that pattern.

在上面的表达式中，如果 theValue 求值是真，则表达式的结果为 theValue，否则结果为 defaultValue。同样，如果 theValue 是除了 undefined 和 null 的假值，也会返回 defaultValue。让我们看看这种模式的三个例子。

**Example 1: a default for a parameter**

**示例1：参数的默认值**

The parameter text of the function saveText() is optional and should be the empty string if it has been omitted:

方法 saveText() 中的参数 text 是可选的，如果缺少该参数，默认是空字符串。

function saveText(text) { text = text || ''; ...

}

This is the most common use of || as a default operator. Consult “Optional Parameters” on page 173 for more on optional parameters.

使用 || 作为默认值操作符是最常见的方式。更多关于可选参数的内容可以翻阅第173页“可选参数”。

**Example 2: a default for a property**

**示例2：属性的默认值**

The object options may or may not have the property title. If it is missing, the value 'Untitled' should be used when setting the title:

options 对象可能有title属性，也可能没有。如果没有，设置title时会使用“Untitled”：

setTitle(options.title || 'Untitled');

**Example 3: a default for the result of a function**

**示例3：方法的默认结果**

The function countOccurrences counts how often regex matches inside str:

countOccurrences 方法 返回 regex 匹配 str 的次数

function countOccurrences(regex, str) {

// Omitted: check that /g is set for `regex`

return (str.match(regex) || []).length;

}

The problem is that match() (see “String.prototype.match: Capture Groups or Return All Matching Substrings” on page 307) either returns an array or null. Thanks to ||, a default value is used in the latter case. Therefore, you can safely access the property length in both cases.

问题是 match() (查看第307页“String.prototype.match:捕获组或返回所有匹配的子字符串”) 可能返回数组，也可能返回null。多亏有 || ，后一种情况可以使用默认值。因此，你可以安全的访问两种情况返回结果的 length 属性。

**Logical Not (!)**

**逻辑非（!）**

The logical not operator ! converts its operand to boolean and then negates it:

逻辑非操作符 ! 把操作数转化为布尔值，然后取反：

> !true

false

> !43

false

> !''

true

> !{}

false

**Equality Operators, Ordering Operators**

**相等操作符，排序操作符**

The following operators are covered elsewhere:

下面的操作符将在其他章节介绍：

* Equality operators: ===, !==, ==, != (see “Equality Operators: === Versus ==” on  page 83)
* 相等操作符：===， !==, ==, != (详见第83页的“相等操作符：=== 对比 == ”)
* Ordering operators: >, >=, <, <= (see “Ordering Operators” on page 87)
* 排序操作符：>, >=,<, <= (详见第87页的“排序操作符”)

**The Function Boolean**

**Boolean 方法**

The function Boolean can be invoked in two ways:

Boolean 方法可以通过两种方式调用：

 Boolean(value)

As a normal function, it converts value to a primitive boolean (see “Converting to Boolean” on page 97):

作为一个普通方法，将 value 转化为 基本布尔类型（详见第97页的“转化为Boolean”）

> Boolean(0)

false

> typeof Boolean(false) // no change

 'boolean'

new Boolean(bool)

As a constructor, it creates a new instance of Boolean (see “Wrapper Objects for Primitives” on page 75), an object that wraps bool (after converting it to a boolean). For example:

作为一个构造函数，创建 Boolean 的一个新实例（详见第75页的“包装基本对象”），把 bool 转化为布尔值后包装成对象。例如：

> typeof new Boolean(false)

 'object'

The former invocation is the common one.

前一种调用方式比较常见。