第4章 强制类型转

1. 值类型转换
2. 将值从一种类型转换为另一种类型通常称为类型转换（type casting），这是显式的情况；隐式的情况称为强制类型转换（coercion）。
3. 类型转换发生在静态类型语言的编译阶段，而强制类型转换则发生在动态类型语言的运行时（runtime）。
4. 抽象值操作
5. ToString（处理非字符串到字符串的强制类型转换）

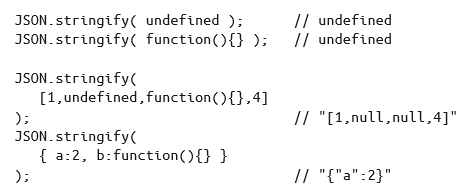
1）基本类型值的字符串化规则为：null 转换为 "null"，undefined 转换为 "undefined"，true 转换为 "true"。

2）对普通对象来说，除非自行定义，否则 toString()（Object.prototype.toString()）返回 内部属性 [[Class]] 的值（参见第 3 章），如 "[object Object]"。

3）toString() 可以被显式调用，或者在需要字符串化时自动调用

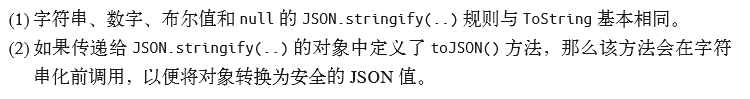
4）JSON 字符串化（JSON字符串化和 toString() 的效果基本相同，只不过序列化的结果总是字符串）

5）JSON.stringify(..) 在对象中遇到 undefined、function 和 symbol 时会自动将其忽略，在 数组中则会返回 null（以保证单元位置不变）。



6）toJSON() 应该“返回一个能够被字符串化的安全的 JSON 值”

7）JSON.stringify(..) 并不是强制类型转换



1. ToNumber
2. true 转换为 1，false 转换为 0。undefined 转换为 NaN，null 转换为 0。
3. 如果 valueOf() 和 toString() 均不返回基本类型值，会产生 TypeError 错误。
4. ToBoolean
5. 假值：假值的布尔强制类型转换结果为 false

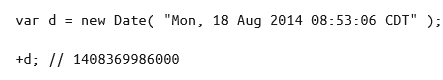
Undefined，null，false，+0，-0，NaN，“ ”

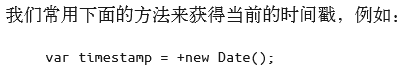
1. 假值对象

假值对象看起来和普通对象并无二致（都有属性，等等），但将它们强制类型转换为布尔值时结果为 false。

1. 真值：真值就是假值列表之外的值。
2. 显式强制类型转换
3. 字符串和数字之间的显式转换
4. 字符串和数字之间的转换是通过 String(..) 和 Number(..) 这两个内建函数（原生构造函数）来实现的，请注意它们前面没有 new 关键字，并不创建封装对象。
5. 日期显式转换为数字

一元运算符 + 的另一个常见用途是将日期（Date）对象强制类型转换为数字，返回结果为 Unix 时间戳，以微秒为单位





将日期对象转换为时间戳并非只有强制类型转换这一种方法，或许使用更显式的方法会更好一些：

var timestamp = new Date().getTime();

1. 奇特的 ~ 运算符

~ 运算符（即字位操作“非”）相关的强制类型转（字位运算符只适用于 32 位整数）

1. 字位截除

使用 ~~ 来截除数字值的小数部分，~ 中的第一个 ~ 执行 ToInt32 并反转字位，然后第二个 ~ 再进行一次字位反转，即将所有 字位反转回原值，最后得到的仍然是 ToInt32 的结果。

1. 显式解析数字字符串

解析允许字符串中含有非数字字符，解析按从左到右的顺序，如果遇到非数字字符就停止。而转换不允许出现非数字字符，否则会失败并返回 NaN。

1. 显式转换为布尔值

Boolean(..)（不带 new）是显式的 ToBoolean 强制类型转换：

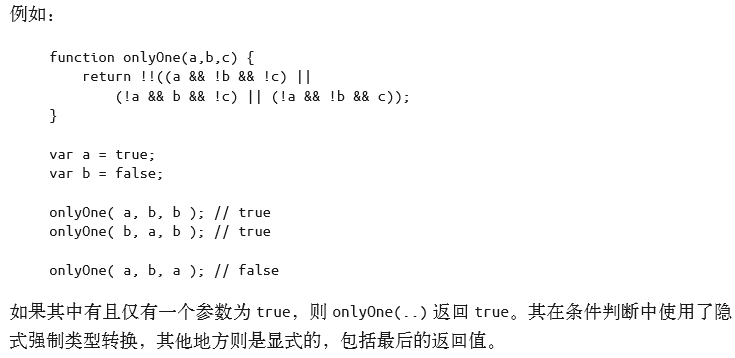
1. 隐式强制类型转换
2. 隐式地简化

隐式强制类型转换同样可以用来提高代码可读性。

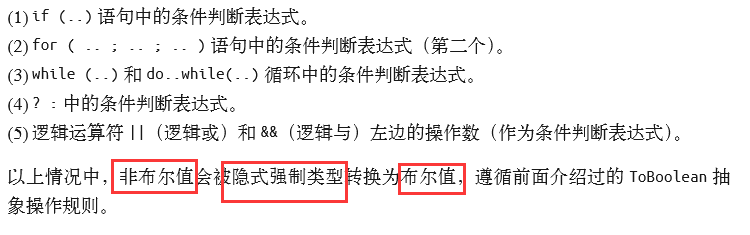
1. 字符串和数字之间的隐式强制类型转换

+ 运算符即能用于数字加法，也能用于字符串拼

1. 布尔值到数字的隐式强制类型转换

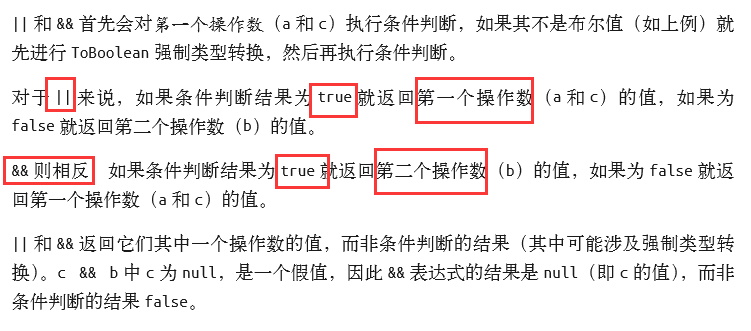


1. 隐式强制类型转换为布尔值



1. || 和 &&

&& 和 || 运算符的返回值并不一定是布尔类型，而是两个操作数其中一个的值。



1. 符号的强制类型转换

符号不能够被强制类型转换为数字（显式和隐式都会产生错误），但可以被强制类型转换为布尔值（显式和隐式结果都是 true）。

1. 宽松相等和严格相等

宽松相等（loose equals）== 和严格相等（strict equals）=== 都用来判断两个值是否“相 等”，但是它们之间有一个很重要的区别，特别是在判断条件上。

1. 相等比较操作的性能

== 和 === 都会检查操作数的类型。区别在于操作数类型不同时它们的处理方 式不同。

1. 抽象相等

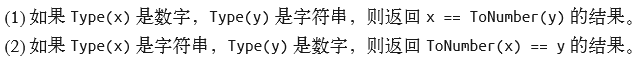
如果两个值的类型相同，就仅比较它们是否相等。例如，42 等于 42，"abc" 等于 "abc"。

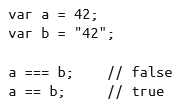


== 在比较两个不同类型的值时会发生隐式强制类型转换，会将其中之 一或两者都转换为相同的类型后再进行比较。

宽松不相等（loose not-equality）!= 就是 == 的相反值，!== 同理

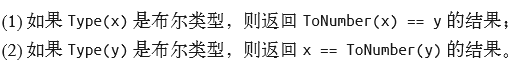
1. 字符串和数字之间的相等比较

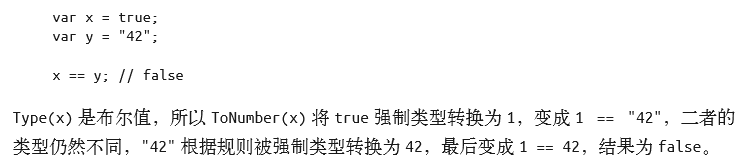




没有强制类型转换，所以 a === b 为 false，42 和 "42" 不相等。

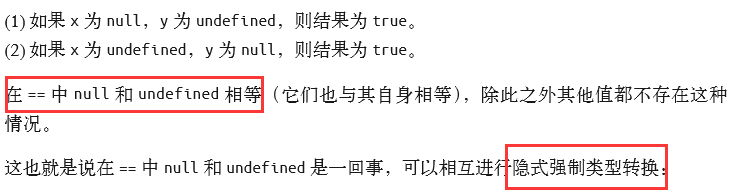
1. 其他类型和布尔类型之间的相等比较



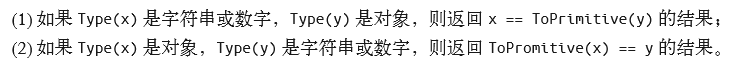


1. null 和 undefined 之间的相等比较

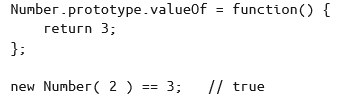
null 和 undefined 之间的 == 也涉及隐式强制类型转换。



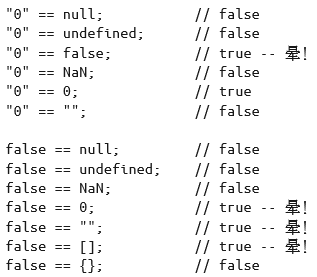
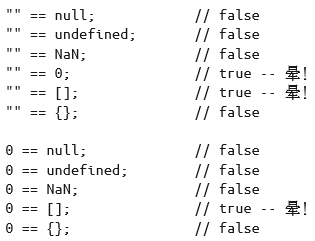
1. 对象和非对象之间的相等比较



1. 比较少见的情况
2. 返回其他数字



1. 假值的相等比较

1. 极端情况

[] == ![] // true

2 == [2]; //

true "" == [null]; // true

== 右边的值 [2] 和 [null] 会进行 ToPrimitive 强制类型转换， 以便能够和左边的基本类型值（2 和 ""）进行比较。因为数组的 valueOf() 返回数组本身， 所以强制类型转换过程中数组会进行字符串化。第一行中的 [2] 会转换为 "2"，然后通过 ToNumber 转换为 2。第二行中的 [null] 会直接转 换为 ""。

0 == "\n"; // true (""、"\n"（或者 " " 等其他空格组合）等空字符串被 ToNumber 强制类型转换 为 0)

1. 完整性检查
2. 安全运用隐式强制类型转换



这时最好用 === 来避免不经意的强制类型转换

1. 抽象关系比较
2. 对 a < b，a=""> b 会被处理为 b <>

比较双方首先调用 ToPrimitive，如果结果出现非字符串，就根据 ToNumber 规则将双方强 制类型转换为数字来进行比较。

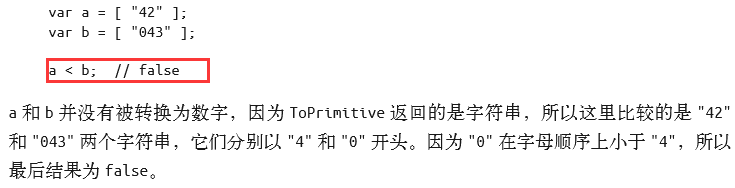
var a = [ 42 ];

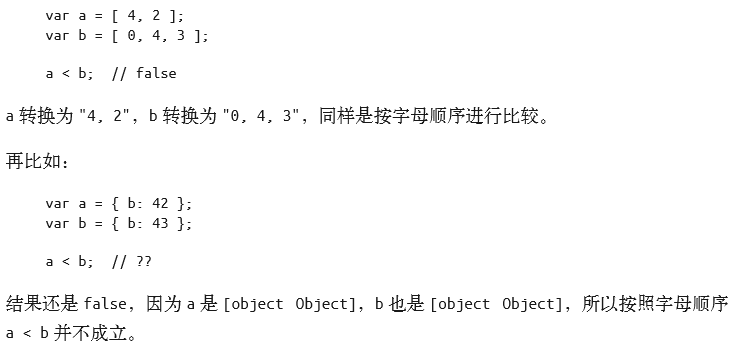
var b = [ "43" ];

a < b; // true

b < a; // false

1. 如果比较双方都是字符串，则按字母顺序来进行比较：





1. 字符串值相同（同为 "[object Object]"

