**一、自动转换为布尔值---Boolean（ if,while,for语句 的条件部分）**

**1）当JavaScript遇到预期为布尔值的地方（比如if语句的条件部分），就会将非布尔值的参数自动转换为布尔值。**

因此除了以下六个值，其他都是自动转为true：

1, undefined

2, null

3, +0

4, -0

5, NaN

6, 空数组

7,空字符串

**二、自动转换为字符串---String**

当JavaScript遇到预期为字符串的地方，就会将非字符串的数据自动转为字符串，转换规则与“强制转换为字符串”相同。

**1）字符串的自动转换，主要发生在 加法运算 时。当一个值为字符串，另一个值为非字符串，则后者转为字符串。**

表达式之中，只要有一个是字符串，则另一个不管是什么类型，都会被自动转化为字符串，然后执行字符串连接运算。通俗说就是：加号（+）两边只要有字符串，就把相应的值转化成字符串，然后做字符串连接操作

Example：

33 + "abcd"  //  "33abcd"

"abcd" + 22   //  "abcd22"

**三、自动转换为数值**

**1)除了加法运算符有可能把数据转为字符串外，其他运算符---减乘除  都会把两侧的数据类型自动转成数值。**

当JavaScript遇到预期为数值的地方，就会将参数值自动转换为数值，转换规则与“强制转换为数值”相同。

ex:

"4" - "2"  //  2

"3" \* "2"  //  6

true - 1  //  0

false -1  //  -1

"11" - 1  //  10

"5" \* []  //  0

false / "5"  //  0

"abc" - 1   //  NaN  ---  "abc"转化成数值是NaN，和NaN做任何数值运算结果都是NaN

\*以上是二元算数运算符的例子，JavaScript的两个一元算数运算符——正号（+）和负号（-）也会把运算子自动转为数值。

Example:

+ "ac"   //  NaN

- "bc"   //  NaN

+ true   //  1

- false  //  0

- "4"  //   -4

**2)表达式为数值或布尔值的时候，布尔值自动转换成数值**

这种情况下，执行加法运算，布尔值转为数值（true为1，fal se为0）。

Example:

true + 7 //    8

true + false   //   1

true + true   //   2

8 - true   //    7

true \* 8    //   8

**四，运算表达式中存在对象**

运算表达式之中若存在对象（或者准确地说，存在非原始类型的值），则先调用该对象的val ueOf（）方法。如果返回结果为原始类型的值，则运用上面的规则；否则继续调用该对象的toString（）方法，对其返回值运用上面两条规则。

Example：

3 + [3,23]    //    "33,23"  ---  先调用[3,23].valueOf()，结果还是数组[3，23]本身，则继续调用[3,23].toString()，结果字符串“3,23”，所以最终结果为字符串“33,23”。

2+ {a:1}     //    "2[object Object]"   ---   先调用{a:1}的valueOf方法，返回的就是这个对象的本身，因此接着对它调用toString方法。({a:1}).toString()默认返回字符串"[object Object]"，所以最终结果就是字符串“1[object Object]”

{ a : 1 } + 2      //    2   ---   JavaScript引擎不将{a:1}视为对象，而是视为一个代码块，这个代码块没有返回值，所以被忽略。因此上面的代码，实际上等同于 {a:1};+2 ，所以最终结果就是2。为了避免这种情况，需要对{a:1}加上括号。如下

( { a : 1 } ) + 2     //     "[object Object]2"    ----   将{a:1}放置在括号之中，由于JavaScript引擎预期括号之中是一个值，所以不把它当作代码块处理，而是当作对象处理，所以最终结果为“[object Object]2”。

**五、四个特殊表达式**

**1)空数组 + 空数组**

    [] + []   //   ""    ---      首先，对空数组调用valueOf方法，返回的是数组本身；因此再对空数组调用toString方法，生成空字符串；所以，最终结果就是空字符串。

**2)空数组 + 空对象**

   [ ]  +  { }    //    "[object Object]"     ---   这等同于空字符串与字符串“[object Object]”相加。因此，结果就是“[object Object]”。

**3)空对象 + 空数组**

  { }  + []      //   0        ---      JavaScript引擎将空对象视为一个空的代码块，加以忽略。因此，整个表达式就

变成“+ []”，等于对空数组求正值，因此结果就是0。

转化过程如下：+   [ ]

//   Number ( [ ] )

//   Number ( [ ] . toString ( ) )

//   Number ( " " )

//   0

如果JavaScript不把前面的空对象视为代码块，则结果为字符串“[object Object]”。

( { } ) + [ ]     //    "[object Object]"

4)空对象 + 空对象

{ }   +   { }    //    NaN     ---      JavaScript同样将第一个空对象视为一个空代码块，整个表达式就变成“+ {}”。这时，后一个空对象的Val ueOf方法得到本身，再调用toSting方法，得到字符串“[object Object]”，然后再将这个字符串转成数值，得到NaN。所以，最后的结果就是NaN。

转化过程如下： +   { }

//  Number( { } )

//  Number( { } . toString ( ) )

//  Number( " [object    Object ] " )

如果，第一个空对象不被JavaScript视为空代码块，就会得到“[object Object][object Object]”的结果。

( { } )   +   { }     / /     " [object Object][object Object] "

( { }   +   { } )    / /      " [object Object][object Object] "

console . log ( { }   +   { } )        / /   " [object Object][object Object]"

var   a   =   { }   +   { } ;      a         / /   "[object Object][object Object] "

需要指出的是，对于第三和第四种情况，Node.j s的运行结果不同于浏览器环境。

{ }   +   { }     / /     " [object Object] [object Object]"

{ }   +   [ ]   / /   " [object Object] "

可以看到，Node.j s没有把第一个空对象视为代码块。原因是Node.j s的命令行环境，内部执行机制大概是下面的样子：

eval . call ( this , " ( function ( ) { return   { }   +   { } } ) . call ( this ) " )

Node.j s把命令行输入都放在eval中执行，所以不会把起首的大括号理解为空代码块加以忽略。