null == 0 的值是什么？为什么？

单纯从这个题目本身来看，没有什么东西，而且这个问题的答案我们通过浏览器就能得到印证。我们今天要说的不是这个问题，而是对于 == 到底是如何运算的？

首先我们来解决这个问题 null == 0 ？结果是什么呢？我们先通过浏览器来验证一下。

```
null == 0
false
```

在这里我们得到的答案，false，是不是今天的文章就结束了呢，显然没有。我们深入的来探究一下，为什么是这样的结果呢。
接下来我们来探究一下 为什么是这样的一个过程！

为了证明一下说明的内容是浏览器 JS 引擎的设置规则，而非个人认为，下面引用 W3C 的制定规范。

以下内容来自 W3C 标准
11.9.3 抽象相等比较法。

比较运算 x==y，其中 x 和 y 是值，产生 true 或者 false。这样的比较按如下方式进行：

若 Type(x)与 Type(y)相同， 则
若 Type(x)为 Undefined， 返回 true。
若 Type(x)为 Null， 返回 true。
若 Type(x)为 Number， 则
若 x 为 NaN， 返回 false。
若 y 为 NaN， 返回 false。
若 x 与 y 为相等数值， 返回 true。
若 x 为 +0 且 y 为-0， 返回 true。
若 x 为 -0 且 y 为+0， 返回 true。
返回 false。
若 Type(x)为 String，则当 x 和 y 为完全相同的字符序列(长度相等且相同字符在相同位置）时返回 true。 否则， 返回 false。
若 Type(x)为 Boolean，当 x 和 y 为同为 true 或者同为 false 时返回 true。 否则， 返回 false。
当 x 和 y 为引用同一对象时返回 true。否则，返回 false。
若 x 为 null 且 y 为 undefined， 返回 true。
若 x 为 undefined 且 y 为 null， 返回 true。

若 Type(x) 为 Number 且 Type(y) 为 String， 返回 comparison x == ToNumber(y)的结果。
若 Type(x) 为 String 且 Type(y)为 Number，
返回比较 ToNumber(x) == y 的结果。
若 Type(x)为 Boolean， 返回比较 ToNumber(x) == y 的结果。
若 Type(y)为 Boolean， 返回比较 x == ToNumber(y)的结果。
若 Type(x) 为 String 或 Number， 且 Type(y) 为 Object， 返回比较 x == ToPrimitive(y)的结果。
若 Type(x)为 Object 且 Type(y)为 String 或 Number，返回比较 ToPrimitive(x) == y 的结果。
不满足上述条件则返回 false。

我们一打眼，看着上面的规则很多。接下来我们根据实际应用层面给大家阐述上面的内容。

这里要说明一下：

这里 Type 函数并不是 typeof，他只能返回基本数据类型(Null Undefined Object Number String Boolean)

其实==的判断规则分为两大类： 类型相同和不相同
1. 类型相同
   以下是相等的内容
   null == null
   undefined == undefined
   +0 == -0
   true == true
   false == false
   对象的引用地址相同则相同
字符串完全相同的字符序列（长度相等且相同字符在相同位置）时返回 true。
数字相同则为 true
上面的这些内容都是规定，不存在隐式类型转换。

不满足以上条件的同类型元素的 == 式就是 false
对于 NaN 来说，只要存在 NaN 就一定是 false。

2. 类型不同
   undefined == null 值为 true 这个是规定，不存在类型转换。

   前提是判断 X == Y 时，在类型不同时，采取的规则

当 X, Y 中任意一人是 Boolean 类型时，把 Boolean 类型的转换成数字类型进行判断。

转化规则 **true == 1 false == 0**
例如：
true == ‘1’
这里先进行 数字化转化得到 true 转化成 1 ，最终比较的是 数字 1 和 字符串的 1。也就是 1 == ‘1’
False == [] 也是一样的根据这条规则转化成 0 == []
在这里大家会经常出现的一个问题?
在我们写代码的过程中可能会产出类似以下的代码

```
if(a == true) {
// coding
}
```

很显然这里在我们判断最终的结果是 a == 1,所以可能和大家的预想不一样。
推荐大家这样使用:

```
// 显式使用
if(a) {
// coding
}
// 强制转换
if(!!a) {
// coding
}
// 强制转换
if(Boolean(a)) {
// coding
}
```


那上面我们保留着一个问题就是 1 == ‘1’ 这个的判断规则是什么啊?

接下来就是 字符串与数字的比较了。

当 X Y 中一个是数字另一个是字符串，比较规则是 把字符转换成数字然后在进行比较。
1 == ‘1’ 就是 把 ‘1’ 转换成数字 1 在进行比较。

最后就差一个 Object 的类型比较了。
这里我们引入原话
若 Type(x) 为 Object 且 Type(y) 为 String 或 Number，返回比较 ToPrimitive(x) == y 的结果。

Type(y) 是 Boolean 类型，在比较过程中也会被转化成 Number。上面描述的没有问题撒。

这里面提到一个 toPrimitive(x),抽象操作，规范中是咋说的捏?

9.1 ToPrimitive 运算符接受一个值，和一个可选的 期望类型 作参数。ToPrimitive 运算符把其值参数转换为非对象类型。如果对象有能力被转换为不止一种原语类型，可以使用可选的 期望类型 来暗示那个类型。返回该对象的默认值。对象的默认值由把期望类型传入作为 hint 参数调用对象的内部方法[[DefaultValue]]得到，[[DefaultValue]]这个内部方法由 8.12.8 定义。
8.12.8
用字符串 hint 调用 O 的 [[DefaultValue]] 内部方法，采用以下步骤:

令 toString 为用参数 "toString" 调用对象 O 的 [[Get]] 内部方法的结果。
如果 IsCallable(toString) 是 true，则
令 str 为用 O 作为 this 值，空参数列表调用 toString 的 [[Call]] 内部方法的结果。
如果 str 是原始值，返回 str。
令 valueOf 为用参数 "valueOf" 调用对象 O 的 [[Get]] 内部方法的结果。
如果 IsCallable(valueOf) 是 true，则
令 val 为用 O 作为 this 值，空参数列表调用 valueOf 的 [[Call]] 内部方法的结果。
如果 val 是原始值，返回 val。
抛出一个 TypeError 异常。
当用数字 hint 调用 O 的 [[DefaultValue]] 内部方法，采用以下步骤:

令 valueOf 为用参数 "valueOf" 调用对象 O 的 [[Get]] 内部方法的结果。
如果 IsCallable(valueOf) 是 true，则
令 val 为用 O 作为 this 值，空参数列表调用 valueOf 的 [[Call]] 内部方法的结果。
如果 val 是原始值，返回 val。
令 toString 为用参数 "toString" 调用对象 O 的 [[Get]] 内部方法的结果。
如果 IsCallable(toString) 是 true，则
令 str 为用 O 作为 this 值，空参数列表调用 toString 的 [[Call]] 内部方法的结果。
如果 str 是原始值，返回 str。
抛出一个 TypeError 异常。
当不用 hint 调用 O 的 [[DefaultValue]] 内部方法时，除非 O 是 Date 对象的情况下把 hint 当作字符串一样解释它的行为，除此之外把 hint 当作数字一样解释它的行为。

上面说明的 [[DefaultValue]] 在原生对象中只能返回原始值。如果一个宿主对象实现了它自身的 [[DefaultValue]] 内部方法，那么必须确保其 [[DefaultValue]] 内部方法只能返回原始值。

一看到这里，哇塞，这也太多了，我也看不懂啊。其实没那么复杂。

如果是对象的话，先检查是否具有 valueOf 函数，如果有并且返回基本类型值，就使用该值进行强制类型转换

```
var obj = {
value: '123',
valueOf() {
console.log("valueOf")
return this.value
```

```
}
}

console.log(obj == 123)
```

结果

```
valueOf
true
```

如果没有 ValueOf，或者 ValueOf 返回的不是原始值。继续寻找 toString 函数

```
var obj = {
    value: '123',
    valueOf() {
        console.log("valueOf")
        return {}
    },
    toString() {
        console.log('toString')
        return this.value
    }
}

console.log(obj == 123)
```

结果为:

```
valueOf
toString
true
```

那如果 toString 方法返回的也不是基本类型怎么办?

```
var obj = {
    value: '123',
    valueOf() {
        console.log("valueOf")
        return {}
    },
    toString() {
        console.log('toString')
        return {}

    }
}
console.log(obj == 123)
```

我们看一下结果

```
console.log(obj == 123)
                ^

TypeError: Cannot convert object to primitive value
```

会报一个错误。所以规范强调，务必返回原始值。

这样的话，对于 == 的抽象规则，回去接着看我们的问题
Null == 0？
发现怎么对都对不上，上面的规则，别忘了最后一条。不符合上面规矩的，一律返回 false
所以 null == 0 的值是 false

总结
对于上面的内容，**除了类型相同不存在隐式类型转换外，是直接比较，还有一个特殊的就是 undefined == null**
其他的内容最终都是转换到数字层面进行比较。这么一说相信，同学们都能很好的掌握==