06 Symbol-ES6新基础类型

更新时间：2019-07-01 14:07:12

> 你若要喜爱你自己的价值，你就得给世界创造价值。
>
> ——歌德

symbol 是 ES6 新增的一种基本数据类型，它和 number、string、boolean、undefined 和 null 是同类型的，object 是引用类型。它用来表示独一无二的值，通过 Symbol 函数生成。

> 本小节代码都是纯JavaScript代码，建议在非TypeScript环境练习，你可以在浏览器开发者工具的控制台里练习。但是因为TypeScript也支持Symbol，所以如果需要特别说明的地方，我们会提示在TypeScript中需要注意的内容。

我们先来看例子：

```
const s = Symbol();
typeof s; // 'symbol'
```

我们使用 Symbol 函数生成了一个 symbol 类型的值 s。

> 注意：Symbol 前面不能加 new 关键字，直接调用即可创建一个独一无二的 symbol 类型的值。

我们可以在使用 Symbol 方法创建 symbol 类型值的时候传入一个参数，这个参数需要是字符串的。如果传入的参数不是字符串，会先调用传入参数的 toString 方法转为字符串。先来看例子：

```
const s1 = Symbol("lison");
const s2 = Symbol("lison");
console.log(s1 === s2); // false
// 补充：这里第三行代码可能会报一个错误：This condition will always return 'false' since the types 'unique symbol' and 'unique symbol' have no overla
p.
// 这是因为编译器检测到这里的s1 === s2始终是false，所以编译器提醒你这代码写的多余，建议你优化。
```

上面这个例子中使用 Symbol 方法创建了两个 symbol 值，方法中都传入了相同的字符串'lison'，但是`s1 === s2`却是 false，这就是我们说的，Symbol 方法会返回一个独一无二的值，这个值和任何一个值都不等，虽然我们传入的标识字符串都是"lison"，但是确实两个不同的值。

你可以理解为我们每一个人都是独一无二的，虽然可以有相同的名字，但是名字只是用来方便我们区分的，名字相同但是人还是不同的。Symbol 方法传入的这个字符串，就是方便我们在控制台或程序中用来区分 symbol 值的。我们可以调用 symbol 值的`toString`方法将它转为字符串：

```
const s1 = Symbol("lison");
console.log(s1.toString()); // 'Symbol(lison)'
```

你可以简单地理解 symbol 值为字符串类型的值，但是它和字符串有很大的区别，它不可以和其他类型的值进行运算，但是可以转为字符串和布尔类型值：

```
let s = Symbol("lison");
console.log(s.toString()); // 'Symbol(lison)'
console.log(Boolean(s)); // true
console.log(!s); // false
```

通过上面的例子可以看出，symbol 类型值和对象相似，本身转为布尔值为 true，取反为 false。

2.3.1 作为属性名

在 ES6 中，对象的属性名支持表达式，所以你可以使用一个变量作为属性名，这对于一些代码的简化很有用处，但是表达式必须放到方括号内：

```
let prop = "name";
const obj = {
  [prop]: "Lison"
};
console.log(obj.name); // 'Lison'
```

了解了这个新特性后，我们接着学习。symbol 值可以作为属性名，因为 symbol 值是独一无二的，所以当它作为属性名时，不会和其他任何属性名重复：

```
let name = Symbol();
let obj = {
  [name]: "lison"
};
console.log(obj); // { Symbol(): 'lison' }
```

你可以看到，打印出来的对象有一个属性名是 symbol 值。如果我们想访问这个属性值，就只能使用 name 这个 symbol 值：

```
console.log(obj[name]); // 'lison'
console.log(obj.name); // undefined
```

通过上面的例子可以看到，我们访问属性名为 symbol 类型值的 name 时，我们不能使用点'.'号访问，因为obj.name 这的 name 实际上是字符串 'name'，这和访问普通字符串类型的属性名一样。你必须使用方括号的形式，这样 obj[name] 这的 name 才是我们定义的 symbol 类型的变量 name，之后我们再访问 obj 的[name]属性就必须使用变量name。

> 等我们后面学到 ES6 的类(Class)的时候，会利用此特性实现私有属性和私有方法。

2.3.2 属性名的遍历

使用 Symbol 类型值作为属性名，这个属性不会被 for…in 遍历到，也不会被 Object.keys()、Object.getOwnPropertyNames()、JSON.stringify() 获取到：

```
const name = Symbol("name");
const obj = {
  [name]: "lison",
  age: 18
};
for (const key in obj) {
  console.log(key);
}
// => 'age'
console.log(Object.keys(obj));
// ['age']
console.log(Object.getOwnPropertyNames(obj));
// ['age']
console.log(JSON.stringify(obj));
// '{ "age": 18 }'
```

虽然这么多方法都无法遍历和访问到 Symbol 类型的属性名，但是 Symbol 类型的属性并不是私有属性。我们可以使用 Object.getOwnPropertySymbols 方法获取对象的所有symbol类型的属性名：

```
const name = Symbol("name");
const obj = {
  [name]: "lison",
  age: 18
};
const SymbolPropNames = Object.getOwnPropertySymbols(obj);
console.log(SymbolPropNames);
// [ Symbol(name) ]
console.log(obj[SymbolPropNames[0]]);
// 'lison'
// 如果最后一行代码这里报错提示：元素隐式具有 "any" 类型，因为类型"{ [name]: string; age: number; }"没有索引签名。 那可能是在tsconfig.json里开启了noImplicitAny。因为这里我们还没有学习接口等高级类型，所以你可以先忽略这个错误，或者关闭noImplicitAny。
```

除了 Object.getOwnPropertySymbols 这个方法，还可以用 ES6 新提供的 Reflect 对象的静态方法 Reflect.ownKeys，它可以返回所有类型的属性名，所以 Symbol 类型的也会返回。

```
const name = Symbol("name");
const obj = {
  [name]: "lison",
  age: 18
};
console.log(Reflect.ownKeys(obj));
// [ 'age', Symbol(name) ]
```

2.3.3 Symbol.for()和 Symbol.keyFor()

Symbol 包含两个静态方法，for 和 keyFor。

(1) Symbol.for()

我们使用 Symbol 方法创建的 symbol 值是独一无二的，每一个值都不和其他任何值相等，我们来看下例子：

```
const s1 = Symbol("lison");
const s2 = Symbol("lison");
const s3 = Symbol.for("lison");
const s4 = Symbol.for("lison");
s3 === s4; // true
s1 === s3; // false
// 这里还是会报错误：This condition will always return 'false' since the types 'unique symbol' and 'unique symbol' have no overlap.还是我们说过的，因为这里的表达式始终是true和false，所以编译器会提示我们。
```

直接使用 Symbol 方法，即便传入的字符串是一样的，创建的 symbol 值也是互不相等的。**而使用 Symbol.for 方法传入字符串，会先检查有没有使用该字符串调用 Symbol.for 方法创建的 symbol 值，如果有，返回该值，如果没有，则使用该字符串新创建一个。**使用该方法创建 symbol 值后会在全局范围进行注册。

> 注意：这个注册的范围包括当前页面和页面中包含的 iframe，以及 service sorker，我们来看个例子：

```
const iframe = document.createElement("iframe");
iframe.src = String(window.location);
document.body.appendChild(iframe);

iframe.contentWindow.Symbol.for("lison") === Symbol.for("lison"); // true
// 注意：如果你在JavaScript环境中这段代码是没有问题的，但是如果在TypeScript开发环境中，可能会报错：类型“Window”上不存在属性“Symbol”。
// 因为这里编译器推断出iframe.contentWindow是Window类型，但是TypeScript的声明文件中，对Window的定义缺少Symbol这个字段，所以会报错，所以你可以这样写：
// (iframe.contentWindow as Window & { Symbol: SymbolConstructor }).Symbol.for("lison") === Symbol.for("lison")
// 这里用到了类型断言和交叉类型，SymbolConstructor是内置的类型。
```

上面这段代码的意思是创建一个 iframe 节点并把它放到 body 中，我们通过这个 iframe 对象的 contentWindow 拿到这个 iframe 的 window 对象，在 iframe.contentWindow 上添加一个值就相当于你在当前页面定义一个全局变量一样，我们看到，在 iframe 中定义的键为'lison'的 symbol 值在和在当前页面定义的键为'lison'的 symbol 值相等，说明它们是同一个值。

(2) Symbol.keyFor()

该方法传入一个 symbol 值，返回该值在全局注册的键名：

```
const sym = Symbol.for("lison");
console.log(Symbol.keyFor(sym)); // 'lison'
```

2.3.4 11 个内置 symbol 值

ES6 提供了 11 个内置的 Symbol 值，指向 JS 内部使用的属性和方法。看到它们第一眼你可能会有疑惑，这些不是 Symbol 对象的一个属性值吗？没错，这些内置的 Symbol 值就是保存在 Symbol 上的，你可以把 Symbol.xxx 看做一个 symbol 值。接下来我们来挨个学习一下：

(1) Symbol.hasInstance

对象的 Symbol.hasInstance 指向一个内部方法，当你给一个对象设置以 Symbol.hasInstance 为属性名的方法后，当其他对象使用 instanceof 判断是否为这个对象的实例时，会调用你定义的这个方法，参数是其他的这个对象，来看例子：

```
const obj = {
  [Symbol.hasInstance](otherObj) {
    console.log(otherObj);
  }
};
console.log({ a: "a" } instanceof obj); // false
// 注意：在TypeScript中这会报错，"instanceof" 表达式的右侧必须属于类型 "any"，或属于可分配给 "Function" 接口类型的类型。
// 是要求你instanceof操作符右侧的值只能是构造函数或者类，或者类型是any类型。这里你可以使用类型断言，将obj改为obj as any
```

可以看到当我们使用 instanceof 判断{ a: 'a' }是否是 obj 创建的实例的时候，Symbol.hasInstance 这个方法被调用了。

(2) Symbol.isConcatSpreadable

这个值是一个可读写布尔值，当一个数组的 Symbol.isConcatSpreadable 设为 true 时，这个数组在数组的 concat 方法中不会被扁平化。我们来看下例子：

```
let arr = [1, 2];
console.log([].concat(arr, [3, 4])); // 打印结果为[1, 2, 3, 4]，length为4
let arr1 = ["a", "b"];
console.log(arr1[Symbol.isConcatSpreadable]); // undefined
arr1[Symbol.isConcatSpreadable] = false;
console.log(arr1[Symbol.isConcatSpreadable]); // false
console.log([].concat(arr1, [3, 4])); // 打印结果如下：
/*
 [ ["a", "b", Symbol(Symbol.isConcatSpreadable): false], 3, 4 ]
 最外层这个数组有三个元素，第一个是一个数组，因为我们设置了arr1[Symbol.isConcatSpreadable] = false
 所以第一个这个数组没有被扁平化，第一个元素这个数组看似是有三个元素，但你在控制台可以看到这个数组的length为2
 Symbol(Symbol.isConcatSpreadable): false不是他的元素，而是他的属性，我们知道数组也是对象，所以我们可以给数组设置属性
 你可以试试如下代码，然后看下打印出来的效果：
  let arr = [1, 2]
  arr.props = 'value'
  console.log(arr)
*/
```

(3) Symbol.species

这里我们需要提前使用类的知识来讲解这个 symbol 值的用法，类的详细内容我们会在后面课程里全面讲解。这个知识你需要在纯JavaScript的开发环境中才能看出效果，你可以在浏览器开发者工具的控制台尝试。在TypeScript中，下面两个例子都是一样的会报a.getName is not a function错误。

首先我们使用 class 定义一个类 C，使用 extends 继承原生构造函数 Array，那么类 C 创建的实例就能继承所有 Array 原型对象上的方法，比如 map、filter 等。我们先来看代码：

```
class C extends Array {
  getName() {
    return "lison";
  }
}
const c = new C(1, 2, 3);
const a = c.map(item => item + 1);
console.log(a); // [2, 3, 4]
console.log(a instanceof C); // true
console.log(a instanceof Array); // true
console.log(a.getName()); // "lison"
```

这个例子中，a 是由 c 通过 map 方法衍生出来的，我们也看到了，a 既是 C 的实例，也是 Array 的实例。但是如果我们想只让衍生的数组是 Array 的实例，就需要用 Symbol.species，我们来看下怎么使用：

```
class C extends Array {
  static get [Symbol.species]() {
    return Array;
  }
  getName() {
    return "lison";
  }
}
const c = new C(1, 2, 3);
const a = c.map(item => item + 1);
console.log(a); // [2, 3, 4]
console.log(a instanceof C); // false
console.log(a instanceof Array); // true
console.log(a.getName()); // error a.getName is not a function
```

就是给类 C 定义一个静态 get 存取器方法，方法名为 Symbol.species，然后在这个方法中返回要构造衍生数组的构造函数。所以最后我们看到，`a instanceof C` 为 false，也就是 a 不再是 C 的实例，也无法调用继承自 C 的方法。

(4) Symbol.match、Symbol.replace、Symbol.search 和 Symbol.split

这个 Symbol.match 值指向一个内部方法，当在字符串 str 上调用 match 方法时，会调用这个方法，来看下例子：

```
let obj = {
  [Symbol.match](string) {
    return string.length;
  }
};
console.log("abcde".match(obj)); // 5
```

相同的还有 Symbol.replace、Symbol.search 和 Symbol.split，使用方法和 Symbol.match 是一样的。

(5) Symbol.iterator

数组的 Symbol.iterator 属性指向该数组的默认遍历器方法：

```
const arr = [1, 2, 3];
const iterator = arr[Symbol.iterator]();
console.log(iterator);
console.log(iterator.next());
console.log(iterator.next());
console.log(iterator.next());
console.log(iterator.next());
```

这个 Symbol.iterator 方法是可写的，我们可以自定义遍历器方法。

(6) Symbol.toPrimitive

对象的这个属性指向一个方法，当这个对象被转为原始类型值时会调用这个方法，这个方法有一个参数，是这个对象被转为的类型，我们来看下：

```
let obj = {
  [Symbol.toPrimitive](type) {
    console.log(type);
  }
};
// const b = obj++ // number
const a = `abc${obj}`; // string
```

(7) Symbol.toStringTag

Symbol.toStringTag 和 Symbol.toPrimitive 相似，对象的这个属性的值可以是一个字符串，也可以是一个存取器 get 方法，当在对象上调用 toString 方法时调用这个方法，返回值将作为"[object xxx]"中 xxx 这个值：

```
let obj = {
  [Symbol.toStringTag]: "lison"
};
obj.toString(); // "[object lison]"
let obj2 = {
  get [Symbol.toStringTag]() {
    return "haha";
  }
};
obj2.toString(); // "[object haha]"
```

(9) Symbol.unscopables

这个值和 with 命令有关，我们先来看下 with 怎么使用：

```
const obj = {
  a: "a",
  b: "b"
};
with (obj) {
  console.log(a); // "a"
  console.log(b); // "b"
}
// 如果是在TypeScript开发环境中，这段代码可能with会报错：不支持 "with" 语句，这是因为在严格模式下，是不允许使用with的。
```

可以看到，使用 with 传入一个对象后，在代码块中访问对象的属性就不需要写对象了，直接就可以用它的属性。对象的 Symbol.unscopables 属性指向一个对象，该对象包含了当使用 with 关键字时，哪些属性被 with 环境过滤掉：

```
console.log(Array.prototype[Symbol.unscopables]);
/*
{
    copyWithin: true
    entries: true
    fill: true
    find: true
    findIndex: true
    includes: true
    keys: true
    values: true
}
*/
```

2.3.5 在TypeScript中使用symbol类型

2.3.5.1 基础

学习完ES6标准中Symbol的所有内容后，我们来看下在TypeScript中使用symbol类型值，很简单。就是制定一个值的类型为symbol类型：

```
let sym: symbol = Symbol()
```

2.3.5.2 unique symbol

TypeScript在2.7版本对Symbol做了补充，增加了**unique symbol**这种类型，他是symbols的子类型，这种类型的值只能由Symbol()或Symbol.for()创建，或者通过指定类型来指定一个值是这种类型。这种类型的值仅可用于常量的定义和用于属性名。另外还有一点要注意，定义unique symbol类型的值，必须用const不能用let。我们来看个在TypeScript中使用Symbol值作为属性名的例子：

```ts
const key1: unique symbol = Symbol()
let key2: symbol = Symbol()
const obj = {
    [key1]: 'value1',
    [key2]: 'value2'
}
console.log(obj[key1])
console.log(obj[key2]) // error 类型“symbol”不能作为索引类型使用。
```

小结

本小节我们详细学习了Symbol的全部知识，本小节的内容较多：我们学习了Symbol值的基本使用，使用Symbol函数创建一个symbol类型值，可以给它传一个字符串参数，来对symbol值做一个区分，但是即使多次Symbol函数调用传入的是相同的字符串，创建的symbol值也是彼此不同的。

我们还学习了Symbol的两个静态方法：Symbol.for和Symbol.keyFor，Symbol.for调用时传入一个字符串，使用此方式创建symbol值时会先在全局范围搜索是否有用此字符串注册的symbol值。如果没有创建一个新的；如果有返回这个symbol值，Symbol.keyFor则是传入一个symbol值然后返回该值在全局注册时的标志字符串。我们还学习了11个内置的symbol值，在设计一些高级逻辑时，可能会用到，大部分业务开发很少用到，你可以了解这些值的用途，日后如果遇到这个需求可以想到这有这些内容。

下个小节我们将对第二个前面大致介绍的知识点——枚举Enum进行详细学习，学完后你将全面了解枚举。

05 TS中补充的六个类型
07 深入学习枚举