**第10章 Symbol**

**10.1 概述**

ES5的对象属性名都是字符串，这容易造成属性名的冲突。比如，你使用了一个他人提供的对象，但又想为这个对象添加新的方法（mixin模式），新方法的名字就有可能与现有方法产生冲突。如果有一种机制，能够保证每个属性的名字都是独一无二的就好了，这样就能从根本上防止属性名冲突。这就是ES6引入Symbol的原因。

ES6引入了一种新的原始数据类型Symbol，表示独一无二的值。它是JavaScript语言的第七种数据类型，前6种分别是：Nudefined、Null、布尔值（Boolean）、字符串（String）、数值（Number）和对象（Object）。

Symbol值通过Symbol函数生成。这就是说，对象的属性名现在可以有两种类型：一种是原来就有的字符串，另一种就是新增的Symbol类型。只要属性名属于Symbol类型，就是独一无二的，可以保证不会与其他属性名产生冲突。

1. let s = Symbol();

2.

3. typeof s;

4. // "Symbol"

5.

上面的代码中，变量S就是一个独一无二的值。typeof运算符的结果表明变量S是Symbol数据类型，而不是字符串之类的其他类型。

注意，Symbol函数前不能使用new命令，否则会报错。这是因为生成的Symbol是一个原始类型的值，不是对象。也就是说，由于Symbol值不是对象，所以不能添加属性。基本上，它是一种类似于字符串的数据类型。

Symbol函数可以接受一个字符串作为参数，表示对Symbol实例的描述，主要是为了在控制台显示或转为字符串时比较容易区分。

1. var s1 = Symbol("foo");

2.

3. var s2 = Symbol("bar");

4.

5. s1 // Symbol(foo)

6. s2 // Symbol(bar)

7.

8. s1.toString() // "Symbol(foo)"

9. s2.toString() // "Symbol(bar)"

10.

上面的代码种，s1和s2是两个Symbol值。如果不加参数，它们在控制台的输出都是Symbol()，不利于区分。有了参数以后，就等于为它们加上了描述，输出时就能够分清到底是哪一个值。

注意，Symbol函数的参数只表示对当前Symbol值得描述，因此相同参数得Symbol函数的返回值是不相等的。

1. // 没有参数的情况

2. var s1 = Symbol();

3. var s2 = Symbol();

4.

5. s1 === s2 // false

6.

7. // 有参数的情况

8. var s1 = Symbol("foo");

9. var s2 = Symbol("foo");

10.

11. s1 === s2 // false

12.

上面的代码中，s1和s2都是Symbol函数的返回值，而且参数相同，但是它们是不相等的。

Symbol值不能与其他类型的值进行运算，否则会报错。

1. var sym = Symbol("My Symbol");

2.

3. "your symbol is " + sym

4. // TypeError: can't convert symbol to string

5. 'your symbol is ${sym}'

6. // TypeError: can't convert symbol to string

7.

但是，Symbol值可以显示转为字符串。

1. var sym = Symbol("My Symbol");

2.

3. "your symbol is " + sym

4. // TypeError: can't convert symbol to string

5. 'your symbol is ${sym}'

6. // TypeError: can't convert symbol to string

7.

8. var sym = Symbol("my school");

9. String(sym) // "Symbol('my scholl')"

10. sym.toString() // "Symbol('my scholl')"

11.

另外，Symbol值也可以转为布尔值，但是不能转为数值。

1. var sym = Symbol();

2. Boolean(sym) // true

3.

4. !sym // false

5.

6. if (sym) {

7.     // ...

8. }

9.

10. Number(sym) // TypeError

11.

12. sym + 2 // TypeError

13.

**10.2 作为属性名的Symbol**

由于每一个Symbol值都是不相等的，这意味着Symbol值可以作为标识符用于对象的属性名，保证不会出现同名的属性。这对于一个对象由多个模块构成的情况非常有用，能防止一个键被不小心改写或覆盖。

1. var mySymbol = Symbol();

2. // 第一种写法

3. var a = {};

4. a[mySymbol] = "Hello";

5.

6. // 第二种写法

7. var a = {

8.     [mySymbol]: "Hello"

9. };

10.

11. // 第三种写法

12. var a = {};

13. Object.defineProperty(a, mySymbol, {

14.     value: "Hello"

15. });

16.

17. // 以上写法都得到同样的结果

18. a[mySymbol] // "Hello"

19.

上面的代码通过方括号结构和Object.defineProperty将对象的属性名指定为一个Symbol值。

注意，Symbol值作为对象属性名时不能使用点运算符。

1. var mySymbol = Symbol();

2.

3. var a = {};

4.

5. a.mySymbol = "Hello";

6.

7. a[mySymbol] // undefined

8. a['mySymbol'] // Hello

9.

10.

上面的代码中，因为点运算符后面总是字符串，所以不会读取mySymbol作为标识名所指代的值，导致a的属性名实际上是一个字符串，而不是一个Symbol值。

同理，在对象内部，使用Symbol值定义属性时，Symbol值必须放在方括号中。

1. let s = Symbol();

2.

3. let obj = {

4.     [s]: function (arg) {

5.         // ...

6.     }

7. };

8.

9. obj[s](123);

10.

上面的代码中，如果s不放在方括号中，该属性的键名就是字符串s，而不是s所代表的Symbol值。

采用增强的对象写法，上面的obj对象可以写的更简洁一些。

1. let obj = {

2.     [s](arg) {

3.         // ...

4.     }

5. };

6.

7.

Symbol类型还可用于定义一组常量，保证这组常量的值都是不相等的。

1. log.levels = {

2.     DEBUG: Symbol("debug"),

3.     INFO: Symbol("info"),

4.     WARN: Symbol("warn")

5. };

6.

7. log(log.levels.DEBUG, "debug message");

8. log(log.levels.INFO, "info message");

9.

下面是另外一个例子。

1. const COLOR\_RED = Symbol();

2. const COLOR\_GREEN = Symbol();

3.

4. function getComplement(color) {

5.   switch (color) {

6.     case COLOR\_RED:

7.       break;

8.     case COLOR\_GREEN:

9.       break;

10.     default:

11.       throw new Error("Undefined color");

12.   }

13. }

14.

15.

常量使用Symbol值最大的好处，就是其他任何值都不可能有相同值了，因此可以保证上面的switch语句会按设计的方式工作。

还有一点需要注意，Symbol值作为属性名时，该属性还是公开属性，不是私有属性。

**10.3 实例：消除魔术字符串**

魔术字符串指的是，在代码中多次出现、与代码形成强耦合的某一个具体的字符串或数值。

风格良好的代码，应该尽量消除魔术字符串，而由含义清晰的变量代替。

1. function getArea(shape, options) {

2.   var area = 0;

3.   switch (shape) {

4.     case "Triangle": // 魔术字符串

5.       area = 0.5 \* options.width \* options.height;

6.       break;

7.     /\*\* more code ... \*/

8.     default:

9.       break;

10.   }

11. }

12.

13. getArea("Triangle", { width: 100, height: 100 }); // 魔术字符串

14.

上面的代码中，字符串“Triangle”就是一个魔术字符串。它多次出现，与代码形成“强耦合”，不利于将来的修改和维护。

常用的消除魔术字符串的方法，就是把它写成一个变量。

1. var shapeType = {

2.     triangle: "Triangle"

3. };

4.

5. function getArea(shape, options) {

6.   var area = 0;

7.   switch (shape) {

8.     case shapeType.triangle:

9.       area = 0.5 \* options.width \* options.height;

10.       break;

11.     /\*\* more code ... \*/

12.     default:

13.       break;

14.   }

15. }

16.

17. getArea(shapeType.triangle, { width: 100, height: 100 });

18.

上面的代码中，我们把“Triangle”写成shapeType对象的triangle属性，这样就消除了强耦合。

如果仔细分析，可以发现shapeType.triangle等于哪个值并不重要，只要确保它不会跟其他shapeType属性的值冲突即可。因此，这里就很适合改用Symbol值。

1. const shapeType = {

2.     triangle: Symbol()

3. };

4.

5.

上面的代码中，除了将shapeType.triangle的值设为一个Symbol，其他地方都不用修改。

**10.4 属性名的遍历**

Symbol作为属性名，该属性不会出现在for…in、for…of循环中，也不会被Object.keys()、Object.getOwnPropertyNames()返回。但它也不是私有属性，有一个Object.getOwnPropertySymbols方法可以获取指定对象的所有Symbol属性名。

Object.getOwnPropertySymbols方法返回一个数组，成员是当前对象的所有用作属性名的Symbol值。

1. var obj = {};

2. 1. var a = Symbol("a");

3. 2. var b = Symbol.for("b");

4. 3.

5. 4. obj[a] = "Hello";

6. 5. obj[b] = "World";

7. 6.

8. 7. var objectSymbols = Object.**getOwnPropertySymbols**(obj);

9. 8.

10. 9. console.log(objectSymbols);

11. 10. // [ Symbol(a), Symbol(b) ]

12. 11.

13.

下面是另一个例子，将Object.getOwnPropertySymbols方法与for…in循环、Object.getOwnPropertyNames方法进行了对比。

1. var obj = {};

2.

3. var foo = Symbol("foo");

4.

5. Object.defineProperty(obj, foo, {

6.   value: "foobar"

7. });

8.

9. for (var i in obj) {

10.   console.log(i); // 无输出

11. }

12.

13. Object.getOwnPropertyNames(obj);

14. // []

15.

16. Object.getOwnPropertySymbols(obj);

17. // [ Symbol(foo) ]

18.

19.

上面的代码中，使用Object.getOwnPropertyNames方法得不到Symbol属性名，需要使用Object.getOwnProperySymbols方法。

另一个新的API---Reflect.ownKeys方法可以返回所有类型的键名，包括常规键名和Symbol键名。

1. let obj = {

2.   [Symbol('my\_key')]: 1,

3.   enum: 2,

4.   nonEnum: 3

5. };

6.

7. Reflect.ownKeys(obj);

8. // [ 'enum', 'nonEnum', Symbol(my\_key) ]

9.

10.

以Symbol值作为名称的属性不会被常规方法遍历得到。我们可以利用这个特性，为对象定义一些非私有但又希望只用于内部的方法。

1. var size = Symbol("size");

2.

3. class Collection {

4.   constructor() {

5.     this[size] = 0;

6.   }

7.

8.   add(item) {

9.     this[this[size]] = item;

10.     this[size]++;

11.   }

12.

13.   static sizeOf(instance) {

14.     return instance[size];

15.   }

16. }

17.

18. var x = new Collection();

19. Collection.sizeOf(x); // 0

20.

21. x.add("foo");

22. Collection.sizeOf(x); // 1

23.

24. Object.keys(x); // ['0']

25. Object.getOwnPropertyNames(x); // ['0']

26. Object.getOwnPropertySymbols(x); // [Symbol(size)]

27.

28.

上面的代码中，对象x的size属性是一个Symbol值，所以Object.keys(x)、Object.getOwnPropertyNames(x)都无法获取它。这就造成了一种非私有的内部方法的效果。

**10.5 Symbol.for()，Symbol.keyFor()**

有时，我们希望重新使用同一个Symbol值，Symbol.for方法可以做到这一点。它接受一个字符串作为参数，然后搜索有没有以该参数作为名称的Symbol值。如果又，就返回这个Symbol值，否则就新建并返回一个以该字符串为名称的Symbol值。

1. var s1 = Symbol.for("xie");

2. var s2 = Symbol.for("xie");

3.

4. s1 === s2 // true

5.

上面的代码中，s1和s2都是Symbol值，但它们都是同样参数的Symbol.for方法生成的，所以实际上是同一个值。

Symbol.for()与Symbol()这两种写法，都会生成新的Symbol。它们的区别是，前者会被登记在全局环境中供搜索，而后者不会。Symbol.for()不会每次调用都返回一个新的Symbol类型的值，而是会先检查给定的key是否已经存在，如果不存在才会新建一个值。比如，如果你调用Symbol.for(“cat”)30次，每次都会返回同一个Symbol值，但是调用Symbol(“cat”)30次，会返回30个不同的Symbol值。

1. Symbol.for("bar") === Symbol.for("bar");

2. // true

3.

4. Symbol("bar") === Symbol("bar")

5. // false

6.

上面的代码中，由于Symbol()写法没有登记机制，所以每次调用都会返回一个不同的值。

Symbol.keyFor方法返回一个已登记的Symbol类型值的key。

1. var s1 = Symbol.Symbol("foo");

2. Symbol.keyFor(s1); // "foo"

3.

4. var s2 = Symbol("foo");

5. Symbol.keyFor(s2); // undefined

6.

上面的代码中，变量s2属于未登记的Symbol值，所以返回undefined。

需要注意的是，Symbol.for为Symbol值登记的名字是全局环境，可以在不同的iframe或service worker中取得同一个值。

1. iframe = document.createElement(“iframe”);

2. iframe.src = String(window.location);

3. document.appenChild(iframe);

4.

5. iframe.contentWindow.Symbol.for(‘foo’) === Symbol.for(“foo”)

6. // true

7.

上面的代码中，iframe窗口生成的Symbol值可以在主页面得到。

**10.6 内置的Symbol值**

除了定义自己使用的Symbol值外，ES6还提供了11个内置的Symbol值，指向语言内部使用的方法。

**Symbol.hasInstance**

对象的Symbol.hasInstance属性指向一个内部方法，对象使用instanceof运算符时会调用这个方法，判断该对象是否为某个构造函数的实例。比如，foo instanceof Foo在语言内部实际调用的是**Foo[Symbol.hasInstance](foo)**。

1. class MyClass {

2.     [Symbol.hasInstance](foo) {

3.         console.log("sss");

4.         return foo instanceof Array;

5.     }

6. }

7.

8. var o = new MyClass();

9. o instanceof Array; // false

10.

**Symbol.isConcatSpredable**

对象的Symbol.isConcatSpreadable属性等于一个布尔值，表示该对象使用Array.prototype.concat()时是否可以展开。

1. let arr1 = ['c', 'd'];

2. ['a', 'b'].concat(arr1, 'e'); // [ 'a', 'b', 'c', 'd', 'e' ]

3.

4. let arr2 = ['c', 'd'];

5. arr2[Symbol.isConcatSpreadable] = false;

6. ['a', 'b'].concat(arr2, 'e');

7. // [

8. //     'a',

9. //     'b',

10. //     [ 'c', 'd', [Symbol(Symbol.isConcatSpreadable)]: false ],

11. //     'e'

12. //   ]

13.

上面的代码说明，数组的Symbol.isConcatSpreadable属性默认为true，表示可以展开。

类似数组的对象也可以展开，但其Symbol.isConcatSpreadable属性默认为false，必须手动打开。

1. let obj = {

2.     length: 2,

3.     0: 'c',

4.     1: 'd'

5. };

6.

7. ['a', 'b'].concat(obj, 'e');

8. // [ 'a', 'b', { '0': 'c', '1': 'd', length: 2 }, 'e' ]

9.

10. obj[Symbol.isConcatSpreadable] = true;

11. ['a', 'b'].concat(obj, 'e');

12. // [ 'a', 'b', 'c', 'd', 'e' ]

13.

14.

对于一个类而言，Symbol.isConcatSpreadable属性必须写成一个返回布尔值的方法。

1. class A1 extends Array {

2.   [Symbol.isConcatSpreadable] = true

3. }

4.

5. class A2 extends Array {

6.   [Symbol.isConcatSpreadable] = false

7. }

8.

9. let a1 = new A1();

10. a1[0] = 3;

11. a1[1] = 4;

12.

13. let a2 = new A2();

14. a2[0] = 5;

15. a2[1] = 6;

16. [1, 2].concat(a1).concat(a2);

17. // [ 1, 2, 3, 4, [ 5, 6 ]]

18. // [

19. //     1,

20. //     2,

21. //     3,

22. //     4,

23. //     A2(2) [ 5, 6, [Symbol(Symbol.isConcatSpreadable)]: false ]

24. // ]

25.

上面的代码中，类A1是可扩展的，类A2是不可扩展的，所以使用concat时有不一样的结果。

**Symbol.species**

对象的Symbol.species属性指向一个方法，对象作为构造函数创造实例时会调用这个方法。即如果this.constructor[Symbol.species]存在，就会使用这个属性作为构造函数来创建新的实例对象。

Symbol.species属性默认的读取器如下。

1. static get [Symbol.species]() {

2. return this;

3. }

4.

**Symbol.match**

对象的Symbol.match属性指向一个函数，当执行str.match(myObject)时，如果该属性存在，会调用它返回该方法的返回值。

1. String.prototype.match(regexp)

2.

3. // 等同于

4. regexp[Symbol.match](this)

5.

6. class MyMatcher {

7.     [Symbol.match](string) {

8.         return "Hello world".indexOf(string);

9.     }

10. }

11.

12. "e".match(new MyMatcher()) // 1

13.

14.

**Symbol.replace**

对象的Symbol.replace属性指向一个方法，当对象被String.prototype.replace方法调用时会返回该方法的返回值。

1. String.prototype.replace(searchValue, replaceValue);

2.

3. // 等同于

4. searchValue[Symbol.replace](this, replaceValue);

5.

6.

**Symbol.search**

对象的symbol.search属性指向一个方法，当对象被String.prototype.search方法调用时会返回该方法的返回值。

1. String.prototype.search(regexp)

2.

3. // 等同于

4. regexp[Symbol.search](this)

5. class MySearch {

6.     constructor(value) {

7.         this.value = value;

8.     }

9.

10.     [Symbol.search](string) {

11.         console.log(string);

12.         return string.indexOf(this.value);

13.     }

14. }

15.

16. "foobar".search(new MySearch("foo")); // 0

17.

18.

**Symbol.split**

对象的Symbol.split属性指向一个方法，对象被String.prototype.split方法调用时会返回该方法的返回值。

1. String.prototype.split(seperator, limit);

2.

3. // 等同于

4. seperator[Symbol.split](this, limit);

5.

**Symbol.iterator**

对象的Symbol.iterator属性指向其默认遍历器方法，即对象在进行for…of循环时会被调用这个方法，返回该对象的默认遍历器，详细介绍参见第14章。

1. class Collection {

2.     **\***[Symbol.iterator]() {

3.         let i = 0;

4.         while(this[i] !== undefined) {

5.             yield this[i];

6.             ++i;

7.         }

8.     }

9. }

10.

11. let myCollection = new Collection();

12.

13. myCollection[0] = 1;

14. myCollection[1] = 2;

15.

16. for (let value of myCollection) {

17.     console.log(value);

18. }

19.

20. // 1

21. // 2

22.

**Symbol.toPrimitive**

对象的Symbol.toPrimitive属性指向一个方法，对象被转为原始类型的值时会调用这个方法，返回该对象对应的原始类型值。

Symbol.toPrimitive被调用时会接受一个字符串参数，表示当前运算的模式。一共有三种模式。

* Number：该场合需要转成数值。
* String：该场合需要转成字符串。
* Default：该场合可用于转成数值，也可以转成字符串。

1. let obj = {

2.   [Symbol.toPrimitive](hint) {

3.     switch (hint) {

4.       case "number":

5.         return 123;

6.       case "string":

7.         return "str";

8.       case "default":

9.         return "default";

10.       default:

11.         return "default";

12.     }

13.   },

14. };

15.

16. 2 \* obj // 246

17. 3 + obj // 3default

18. obj === 'default' // true

19. String(obj) // str

20.

**Symbol.toStringTag**

对象的Symbol.toStringTag属性指向一个方法，在对象上调用Object.prototype.toString方法时，如果**这个属性存在**，其返回值会出现在toString方法返回的字符串中，对象对象的类型。也就是说，这个属性可以用于定制**[Object Object]**或**[Object Array]**中Object后面的字符串。

1. ({[Symbol.toStringTag]: 'Foo'}.toString())

2. // [object Foo]

3.

4. class Collection {

5.     **get** [Symbol.toStringTag]() {

6.         return 'XXX';

7.     }

8. }

9.

10. var x = new Collection();

11.

12. console.log(x.toString()); // [object XXX]

13. console.log(Object.prototype.toString.call(x)); // [object XXX]

14.

ES6新增内置对象的Symbol.toStringTag属性值如下。

* JSON[Symbol.toStringTag]：’JSON’
* Math[Symbol.toStringTag]：’Math’
* Module对象M[Symbol.toStringTag]：’Module’
* ArrayBuffer.prototype[Symbol.toStringTag]：’ArrayBuffer’
* DataView.prototype[Symbol.toStringTag]：’DataView’
* Map.prototype[Symbol.toStringTag]：’Map’
* Promise.prototype[Symbol.toStringTag]：’Promise’
* Set.prototype[Symbol.toStringTag]：’Set‘
* %TypedArray%.prototype[Symbol.toStringTag]：’Uint8Array’等
* WeakSet.prototype[Symbol.toStringTag]：’WeakSet’
* WeakMap.prototype[Symbol.toStringTag]：’WeakMap’
* %MapIteratorPrototype%[Symbol.toStringTag]：’Map Iterator’
* %SetIteratorPrototype%[Symbol.toStringTag]：’Set Iterator’
* %StringIteratorPrototype%[Symbol.toStringTag]：’ String Iterator’
* Symbol.prototype[Symbol.toStringTag]：’Symbol’
* Generator.prototype[Symbol.toStringTag]：’Generator’
* GeneratorFunction.prototype[Symbol.toStringTag]：’GeneratorFunction’

**Symbol.unscopables**

对象的Symbol.unscopables属性指向了一个对象，指定了使用with关键字时哪些属性会被with环境排除。

1. Array.prototype[Symbol.unscopables]

2. // {

3. //     copyWithin: true,

4. //     entries: true,

5. //     fill: true,

6. //     find: true,

7. //     findIndex: true,

8. //     flat: true,

9. //     flatMap: true,

10. //     includes: true,

11. //     keys: true,

12. //     values: true

13. //   }

14.

15. Object.keys(Array.prototype[Symbol.unscopables])

16. // ['copyWithin', 'entries', 'fill', 'find', 'findIndex', 'flat', 'flatMap', 'includes', 'keys', 'values']

17.

18.

上面的代码说明，数组有6个属性会被with命令排除。

1. // // 没有unscopables时

2. class MyClass {

3.     foo() {

4.         return 1;

5.     }

6. }

7.

8. var foo = function () {

9.     return 2;

10. }

11.

12. with (MyClass.prototype) {

13.     foo(); // 1

14. }

15.

16. // 有unscopables

17. class MyClass {

18.     foo() {

19.         return 1;

20.     }

21.     get [Symbol.unscopables]() {

22.         return { foo: true };

23.     }

24. }

25.

26. var foo = function () {

27.     return 2;

28. }

29.

30. with (MyClass.prototype) {

31.

32.     foo(); // 2

33. }

34.