转换为字符串。此外，注册表中使用的键同时也会被用作符号描述。

```
let emptyGlobalSymbol = Symbol.for();
console.log(emptyGlobalSymbol);    // Symbol(undefined)
```

还可以使用 Symbol.keyFor()来查询全局注册表，这个方法接收符号，返回该全局符号对应的字符串键。如果查询的不是全局符号，则返回 undefined。

```
// 创建全局符号
let s = Symbol.for('foo');
console.log(Symbol.keyFor(s));   // foo

// 创建普通符号
let s2 = Symbol('bar');
console.log(Symbol.keyFor(s2));  // undefined
```

如果传给 Symbol.keyFor()的不是符号，则该方法抛出 TypeError：

```
Symbol.keyFor(123); // TypeError: 123 is not a symbol
```

3. 使用符号作为属性

凡是可以使用字符串或数值作为属性的地方，都可以使用符号。这就包括了对象字面量属性和 Object.defineProperty()/Object.defineProperties()定义的属性。对象字面量只能在计算属性语法中使用符号作为属性。

```
let s1 = Symbol('foo'),
    s2 = Symbol('bar'),
    s3 = Symbol('baz'),
    s4 = Symbol('qux');

let o = {
  [s1]: 'foo val'
};
// 这样也可以：o[s1] = 'foo val';

console.log(o);
// {Symbol(foo): foo val}

Object.defineProperty(o, s2, {value: 'bar val'});

console.log(o);
// {Symbol(foo): foo val, Symbol(bar): bar val}

Object.defineProperties(o, {
  [s3]: {value: 'baz val'},
  [s4]: {value: 'qux val'}
});

console.log(o);
// {Symbol(foo): foo val, Symbol(bar): bar val,
//  Symbol(baz): baz val, Symbol(qux): qux val}
```

类似于 Object.getOwnPropertyNames()返回对象实例的常规属性数组，Object.getOwnPropertySymbols()返回对象实例的符号属性数组。这两个方法的返回值彼此互斥。Object.getOwnPropertyDescriptors()会返回同时包含常规和符号属性描述符的对象。Reflect.ownKeys()会返回两种类型的键：

```
let s1 = Symbol('foo'),
    s2 = Symbol('bar');

let o = {
  [s1]: 'foo val',
  [s2]: 'bar val',
  baz: 'baz val',
  qux: 'qux val'
};

console.log(Object.getOwnPropertySymbols(o));
// [Symbol(foo), Symbol(bar)]

console.log(Object.getOwnPropertyNames(o));
// ["baz", "qux"]

console.log(Object.getOwnPropertyDescriptors(o));
// {baz: {...}, qux: {...}, Symbol(foo): {...}, Symbol(bar): {...}}

console.log(Reflect.ownKeys(o));
// ["baz", "qux", Symbol(foo), Symbol(bar)]
```

因为符号属性是对内存中符号的一个引用，所以直接创建并用作属性的符号不会丢失。但是，如果没有显式地保存对这些属性的引用，那么必须遍历对象的所有符号属性才能找到相应的属性键：

```
let o = {
  [Symbol('foo')]: 'foo val',
  [Symbol('bar')]: 'bar val'
};

console.log(o);
// {Symbol(foo): "foo val", Symbol(bar): "bar val"}

let barSymbol = Object.getOwnPropertySymbols(o)
              .find((symbol) => symbol.toString().match(/bar/));

console.log(barSymbol);
// Symbol(bar)
```

4. 常用内置符号

ECMAScript 6 也引入了一批**常用内置符号**（well-known symbol），用于暴露语言内部行为，开发者可以直接访问、重写或模拟这些行为。这些内置符号都以 `Symbol` 工厂函数字符串属性的形式存在。

这些内置符号最重要的用途之一是重新定义它们，从而改变原生结构的行为。比如，我们知道 `for-of` 循环会在相关对象上使用 `Symbol.iterator` 属性，那么就可以通过在自定义对象上重新定义 `Symbol.iterator` 的值，来改变 `for-of` 在迭代该对象时的行为。

这些内置符号也没有什么特别之处，它们就是全局函数 `Symbol` 的普通字符串属性，指向一个符号的实例。所有内置符号属性都是不可写、不可枚举、不可配置的。

> **注意** 在提到 ECMAScript 规范时，经常会引用符号在规范中的名称，前缀为@@。比如，@@iterator 指的就是 Symbol.iterator。

5. Symbol.asyncIterator

根据 ECMAScript 规范，这个符号作为一个属性表示“一个方法，该方法返回对象默认的 AsyncIterator。由 for-await-of 语句使用”。换句话说，这个符号表示实现异步迭代器 API 的函数。

for-await-of 循环会利用这个函数执行异步迭代操作。循环时，它们会调用以 Symbol.asyncIterator 为键的函数，并期望这个函数会返回一个实现迭代器 API 的对象。很多时候，返回的对象是实现该 API 的 AsyncGenerator：

```
class Foo {
  async *[Symbol.asyncIterator]() {}
}

let f = new Foo();

console.log(f[Symbol.asyncIterator]());
// AsyncGenerator {<suspended>}
```

技术上，这个由 Symbol.asyncIterator 函数生成的对象应该通过其 next()方法陆续返回 Promise 实例。可以通过显式地调用 next()方法返回，也可以隐式地通过异步生成器函数返回：

```
class Emitter {
  constructor(max) {
    this.max = max;
    this.asyncIdx = 0;
  }

  async *[Symbol.asyncIterator]() {
    while(this.asyncIdx < this.max) {
      yield new Promise((resolve) => resolve(this.asyncIdx++));
    }
  }
}

async function asyncCount() {
  let emitter = new Emitter(5);

  for await(const x of emitter) {
    console.log(x);
  }
}

asyncCount();
// 0
// 1
// 2
// 3
// 4
```

> **注意**　Symbol.asyncIterator 是 ES2018 规范定义的，因此只有版本非常新的浏览器支持它。关于异步迭代和 for-await-of 循环的细节，参见附录 A。

6. Symbol.hasInstance

根据 ECMAScript 规范，这个符号作为一个属性表示“一个方法，该方法决定一个构造器对象是否

认可一个对象是它的实例。由 instanceof 操作符使用”。instanceof 操作符可以用来确定一个对象实例的原型链上是否有原型。instanceof 的典型使用场景如下：

```
function Foo() {}
let f = new Foo();
console.log(f instanceof Foo); // true

class Bar {}
let b = new Bar();
console.log(b instanceof Bar); // true
```


在 ES6 中，instanceof 操作符会使用 Symbol.hasInstance 函数来确定关系。以 Symbol.hasInstance 为键的函数会执行同样的操作，只是操作数对调了一下：

```
function Foo() {}
let f = new Foo();
console.log(Foo[Symbol.hasInstance](f)); // true

class Bar {}
let b = new Bar();
console.log(Bar[Symbol.hasInstance](b)); // true
```

这个属性定义在 Function 的原型上，因此默认在所有函数和类上都可以调用。由于 instanceof 操作符会在原型链上寻找这个属性定义，就跟在原型链上寻找其他属性一样，因此可以在继承的类上通过静态方法重新定义这个函数：

```
class Bar {}
class Baz extends Bar {
  static [Symbol.hasInstance]() {
    return false;
  }
}

let b = new Baz();
console.log(Bar[Symbol.hasInstance](b)); // true
console.log(b instanceof Bar);           // true
console.log(Baz[Symbol.hasInstance](b)); // false
console.log(b instanceof Baz);           // false
```

7. **Symbol.isConcatSpreadable**

根据 ECMAScript 规范，这个符号作为一个属性表示 “一个布尔值，如果是 true，则意味着对象应该用 Array.prototype.concat()打平其数组元素”。ES6 中的 Array.prototype.concat()方法会根据接收到的对象类型选择如何将一个类数组对象拼接成数组实例。覆盖 Symbol.isConcatSpreadable 的值可以修改这个行为。

数组对象默认情况下会被打平到已有的数组，false 或假值会导致整个对象被追加到数组末尾。类数组对象默认情况下会被追加到数组末尾，true 或真值会导致这个类数组对象被打平到数组实例。其他不是类数组对象的对象在 Symbol.isConcatSpreadable 被设置为 true 的情况下将被忽略。

```
let initial = ['foo'];

let array = ['bar'];
console.log(array[Symbol.isConcatSpreadable]);  // undefined
console.log(initial.concat(array));             // ['foo', 'bar']
array[Symbol.isConcatSpreadable] = false;
console.log(initial.concat(array));             // ['foo', Array(1)]
```

```
let arrayLikeObject = { length: 1, 0: 'baz' };
console.log(arrayLikeObject[Symbol.isConcatSpreadable]);  // undefined
console.log(initial.concat(arrayLikeObject));             // ['foo', {...}]
arrayLikeObject[Symbol.isConcatSpreadable] = true;
console.log(initial.concat(arrayLikeObject));             // ['foo', 'baz']

let otherObject = new Set().add('qux');
console.log(otherObject[Symbol.isConcatSpreadable]);  // undefined
console.log(initial.concat(otherObject));             // ['foo', Set(1)]
otherObject[Symbol.isConcatSpreadable] = true;
console.log(initial.concat(otherObject));             // ['foo']
```

8. Symbol.iterator

根据 ECMAScript 规范，这个符号作为一个属性表示“一个方法，该方法返回对象默认的迭代器。由 for-of 语句使用”。换句话说，这个符号表示实现迭代器 API 的函数。

for-of 循环这样的语言结构会利用这个函数执行迭代操作。循环时，它们会调用以 Symbol.iterator 为键的函数，并默认这个函数会返回一个实现迭代器 API 的对象。很多时候，返回的对象是实现该 API 的 Generator：

```
class Foo {
  *[Symbol.iterator]() {}
}

let f = new Foo();

console.log(f[Symbol.iterator]());
// Generator {<suspended>}
```

技术上，这个由 Symbol.iterator 函数生成的对象应该通过其 next()方法陆续返回值。可以通过显式地调用 next()方法返回，也可以隐式地通过生成器函数返回：

```
class Emitter {
  constructor(max) {
    this.max = max;
    this.idx = 0;
  }

  *[Symbol.iterator]() {
    while(this.idx < this.max) {
      yield this.idx++;
    }
  }
}

function count() {
  let emitter = new Emitter(5);

  for (const x of emitter) {
    console.log(x);
  }
}

count();
// 0
```

```
// 1
// 2
// 3
// 4
```

> **注意** 迭代器的相关内容将在第 7 章详细介绍。

9. Symbol.match

根据 ECMAScript 规范，这个符号作为一个属性表示“一个正则表达式方法，该方法用正则表达式去匹配字符串。由 String.prototype.match()方法使用”。String.prototype.match()方法会使用以 Symbol.match 为键的函数来对正则表达式求值。正则表达式的原型上默认有这个函数的定义，因此所有正则表达式实例默认是这个 String 方法的有效参数：

```
console.log(RegExp.prototype[Symbol.match]);
// ƒ [Symbol.match]() { [native code] }

console.log('foobar'.match(/bar/));
// ["bar", index: 3, input: "foobar", groups: undefined]
```

给这个方法传入非正则表达式值会导致该值被转换为 RegExp 对象。如果想改变这种行为，让方法直接使用参数，则可以重新定义 Symbol.match 函数以取代默认对正则表达式求值的行为，从而让 match()方法使用非正则表达式实例。Symbol.match 函数接收一个参数，就是调用 match()方法的字符串实例。返回的值没有限制：

```
class FooMatcher {
  static [Symbol.match](target) {
    return target.includes('foo');
  }
}

console.log('foobar'.match(FooMatcher)); // true
console.log('barbaz'.match(FooMatcher)); // false

class StringMatcher {
  constructor(str) {
    this.str = str;
  }

  [Symbol.match](target) {
    return target.includes(this.str);
  }
}

console.log('foobar'.match(new StringMatcher('foo'))); // true
console.log('barbaz'.match(new StringMatcher('qux'))); // false
```

10. Symbol.replace

根据 ECMAScript 规范，这个符号作为一个属性表示“一个正则表达式方法，该方法替换一个字符串中匹配的子串。由 String.prototype.replace()方法使用”。String.prototype.replace()方法会使用以 Symbol.replace 为键的函数来对正则表达式求值。正则表达式的原型上默认有这个函数的定义，因此所有正则表达式实例默认是这个 String 方法的有效参数：

```
console.log(RegExp.prototype[Symbol.replace]);
// ƒ [Symbol.replace]() { [native code] }

console.log('foobarbaz'.replace(/bar/, 'qux'));
// 'fooquxbaz'
```

给这个方法传入非正则表达式值会导致该值被转换为 RegExp 对象。如果想改变这种行为，让方法直接使用参数，可以重新定义 Symbol.replace 函数以取代默认对正则表达式求值的行为，从而让 replace()方法使用非正则表达式实例。Symbol.replace 函数接收两个参数，即调用 replace()方法的字符串实例和替换字符串。返回的值没有限制：

```
class FooReplacer {
  static [Symbol.replace](target, replacement) {
    return target.split('foo').join(replacement);
  }
}

console.log('barfoobaz'.replace(FooReplacer, 'qux'));
// "barquxbaz"

class StringReplacer {
  constructor(str) {
    this.str = str;
  }

  [Symbol.replace](target, replacement) {
    return target.split(this.str).join(replacement);
  }
}

console.log('barfoobaz'.replace(new StringReplacer('foo'), 'qux'));
// "barquxbaz"
```

11. Symbol.search

根据 ECMAScript 规范，这个符号作为一个属性表示“一个正则表达式方法，该方法返回字符串中匹配正则表达式的索引。由 String.prototype.search()方法使用”。String.prototype.search()方法会使用以 Symbol.search 为键的函数来对正则表达式求值。正则表达式的原型上默认有这个函数的定义，因此所有正则表达式实例默认是这个 String 方法的有效参数：

```
console.log(RegExp.prototype[Symbol.search]);
// ƒ [Symbol.search]() { [native code] }

console.log('foobar'.search(/bar/));
// 3
```

给这个方法传入非正则表达式值会导致该值被转换为 RegExp 对象。如果想改变这种行为，让方法直接使用参数，可以重新定义 Symbol.search 函数以取代默认对正则表达式求值的行为，从而让 search()方法使用非正则表达式实例。Symbol.search 函数接收一个参数，就是调用 match()方法的字符串实例。返回的值没有限制：

```
class FooSearcher {
  static [Symbol.search](target) {
    return target.indexOf('foo');
  }
}
```

```
console.log('foobar'.search(FooSearcher)); // 0
console.log('barfoo'.search(FooSearcher)); // 3
console.log('barbaz'.search(FooSearcher)); // -1

class StringSearcher {
  constructor(str) {
    this.str = str;
  }

  [Symbol.search](target) {
    return target.indexOf(this.str);
  }
}

console.log('foobar'.search(new StringSearcher('foo'))); // 0
console.log('barfoo'.search(new StringSearcher('foo'))); // 3
console.log('barbaz'.search(new StringSearcher('qux'))); // -1
```


12. Symbol.species

根据 ECMAScript 规范，这个符号作为一个属性表示“一个函数值，该函数作为创建派生对象的构造函数”。这个属性在内置类型中最常用，用于对内置类型实例方法的返回值暴露实例化派生对象的方法。用 `Symbol.species` 定义静态的获取器（getter）方法，可以覆盖新创建实例的原型定义：

```
class Bar extends Array {}
class Baz extends Array {
  static get [Symbol.species]() {
    return Array;
  }
}

let bar = new Bar();
console.log(bar instanceof Array); // true
console.log(bar instanceof Bar);   // true
bar = bar.concat('bar');
console.log(bar instanceof Array); // true
console.log(bar instanceof Bar);   // true

let baz = new Baz();
console.log(baz instanceof Array); // true
console.log(baz instanceof Baz);   // true
baz = baz.concat('baz');
console.log(baz instanceof Array); // true
console.log(baz instanceof Baz);   // false
```

13. Symbol.split

根据 ECMAScript 规范，这个符号作为一个属性表示“一个正则表达式方法，该方法在匹配正则表达式的索引位置拆分字符串。由 `String.prototype.split()` 方法使用”。`String.prototype.split()` 方法会使用以 `Symbol.split` 为键的函数来对正则表达式求值。正则表达式的原型上默认有这个函数的定义，因此所有正则表达式实例默认是这个 `String` 方法的有效参数：

```
console.log(RegExp.prototype[Symbol.split]);
// ƒ [Symbol.split]() { [native code] }

console.log('foobarbaz'.split(/bar/));
// ['foo', 'baz']
```

给这个方法传入非正则表达式值会导致该值被转换为 RegExp 对象。如果想改变这种行为，让方法直接使用参数，可以重新定义 Symbol.split 函数以取代默认对正则表达式求值的行为，从而让 split() 方法使用非正则表达式实例。Symbol.split 函数接收一个参数，就是调用 match()方法的字符串实例。返回的值没有限制：

```
class FooSplitter {
  static [Symbol.split](target) {
    return target.split('foo');
  }
}

console.log('barfoobaz'.split(FooSplitter));
// ["bar", "baz"]

class StringSplitter {
  constructor(str) {
    this.str = str;
  }

  [Symbol.split](target) {
    return target.split(this.str);
  }
}

console.log('barfoobaz'.split(new StringSplitter('foo')));
// ["bar", "baz"]
```

14. Symbol.toPrimitive

根据 ECMAScript 规范，这个符号作为一个属性表示“一个方法，该方法将对象转换为相应的原始值。由 ToPrimitive 抽象操作使用”。很多内置操作都会尝试强制将对象转换为原始值，包括字符串、数值和未指定的原始类型。对于一个自定义对象实例，通过在这个实例的 Symbol.toPrimitive 属性上定义一个函数可以改变默认行为。

根据提供给这个函数的参数（string、number 或 default），可以控制返回的原始值：

```
class Foo {}
let foo = new Foo();

console.log(3 + foo);       // "3[object Object]"
console.log(3 - foo);       // NaN
console.log(String(foo));   // "[object Object]"

class Bar {
  constructor() {
    this[Symbol.toPrimitive] = function(hint) {
      switch (hint) {
        case 'number':
          return 3;
        case 'string':
          return 'string bar';
        case 'default':
        default:
          return 'default bar';
      }
    }
  }
}
```

```
let bar = new Bar();

console.log(3 + bar);      // "3default bar"
console.log(3 - bar);      // 0
console.log(String(bar)); // "string bar"
```

15. Symbol.toStringTag

根据 ECMAScript 规范，这个符号作为一个属性表示“一个字符串，该字符串用于创建对象的默认字符串描述。由内置方法 Object.prototype.toString()使用”。

通过 toString()方法获取对象标识时，会检索由 Symbol.toStringTag 指定的实例标识符，默认为"Object"。内置类型已经指定了这个值，但自定义类实例还需要明确定义：

```
let s = new Set();

console.log(s);                         // Set(0) {}
console.log(s.toString());              // [object Set]
console.log(s[Symbol.toStringTag]);  // Set

class Foo {}
let foo = new Foo();

console.log(foo);                          // Foo {}
console.log(foo.toString());               // [object Object]
console.log(foo[Symbol.toStringTag]);  // undefined

class Bar {
  constructor() {
    this[Symbol.toStringTag] = 'Bar';
  }
}
let bar = new Bar();

console.log(bar);                          // Bar {}
console.log(bar.toString());               // [object Bar]
console.log(bar[Symbol.toStringTag]);  // Bar
```

16. Symbol.unscopables

根据 ECMAScript 规范，这个符号作为一个属性表示“一个对象，该对象所有的以及继承的属性，都会从关联对象的 with 环境绑定中排除”。设置这个符号并让其映射对应属性的键值为 true，就可以阻止该属性出现在 with 环境绑定中，如下例所示：

```
let o = { foo: 'bar' };

with (o) {
  console.log(foo); // bar
}

o[Symbol.unscopables] = {
  foo: true
};

with (o) {
  console.log(foo); // ReferenceError
}
```

> **注意** 不推荐使用 with，因此也不推荐使用 Symbol.unscopables。