为何要使用 Symbol

作为最新的基本类型，Symbol 为 JavaScript 语言带来了很多好处，特别是当其用在对象属性上时。但是，相比较于 String 类型，Symbol 有哪些 String 没有的功能呢？

在深入探讨 Symbol 之前，让我们先看看一些许多开发人员可能都不知道的 JavaScript 特性。

背景

JavaScript 中有两种数据类型：基本数据类型和对象（对象也包括函数），基本数据类型包括简单数据类型，比如 number（从整数到浮点数，从 Infinity 到 NaN 都属于 Number 类型）、boolean、string、undefined、null（注意尽管 typeof null === 'object'，null 仍然是一个基本数据类型）。

基本数据类型的值是不可以改变的，即不能更改变量的原始值。当然**可以**重新对变量进行赋值。例如，代码 let x = 1; x++;，虽然你通过重新赋值改变了变量 x 的值，但是变量的原始值 1 仍没有被改变。

一些语言，比如 C 语言，有按引用传递和按值传递的概念。JavaScript 也有类似的概念，它是根据传递数据的类型推断出来的。如果将值传入一个函数，则在函数中重新对它赋值不会修改它在调用位置的值。但是，如果你**修改**的是基本数据的值，那么修改后的值**会**在调用它的地方被修改。

考虑下面的例子：

```
function primitiveMutator(val) {
  val = val + 1;
}

let x = 1;
primitiveMutator(x);
console.log(x); // 1

function objectMutator(val) {
  val.prop = val.prop + 1;
}

let obj = { prop: 1 };
objectMutator(obj);
console.log(obj.prop); // 2
```

基本数据类型（NaN 除外）总是与另一个具有相同值的基本数据类型完全相等。如下：

```
const first = "abc" + "def";
const second = "ab" + "cd" + "ef";

console.log(first === second); // true
```

然而，构造两个值相同的非基本数据类型则得到**不相等**的结果。我们可以看到发生了什么：

```
const obj1 = { name: "Intrinsic" };
const obj2 = { name: "Intrinsic" };

console.log(obj1 === obj2); // false

// 但是，当两者的 .name 属性为基本数据类型时 console.log(obj1.name === obj2.name); // true
```

对象在 JavaScript 中扮演着重要的角色，几乎**所有地方**可以见到它们的身影。对象通常是键/值对的集合，然而这种形式的最大限制是：对象的键只能是字符串，直到 Symbol 出现这一限制才得到解决。如果我们使用非字符串的值作为对象的键，该值会被强制转换成字符串。在下面的程序中可以看到这种强制转换：

```
const obj = {};
obj.foo = 'foo';
obj['bar'] = 'bar';
obj[2] = 2;
obj[{}] = 'someobj';

console.log(obj);
// { '2': 2, foo: 'foo', bar: 'bar','[object Object]': 'someobj' }
```

注意：虽然有些离题，但是需要知道的是创建 Map 数据结构的部分原因是为了在键不是字符串的情况下允许键/值方式存储。

Symbol 是什么？

现在既然我们已经知道了基本数据类型是什么，也就终于可以定义 Symbol。Symbol 是不能被重新创建的基本数据类型。在这种情况下，Symbol 类似于对象，因为对象创建多个实例也将导致不完全相等的值。但是，Symbol 也是基本数据类型，因为它不能被改变。下面是 Symbol 用法的一个例子：

```
const s1 = Symbol();
const s2 = Symbol();

console.log(s1 === s2); // false
```

当实例化一个 symbol 值时，有一个可选的首选参数，你可以赋值一个字符串。此值用于调试代码，不会真正影响 symbol 本身。

```
const s1 = Symbol('debug');
const str = 'debug';
const s2 = Symbol('xxyy');

console.log(s1 === str); // false
console.log(s1 === s2); // false
console.log(s1); // Symbol(debug)
```

Symbol 作为对象属性

symbols 还有另一个重要的用法，它们可以被当作对象中的键！下面是一个在对象中使用 symbol 作为键的例子：

```
const obj = {};
const sym = Symbol();
obj[sym] = 'foo';
obj.bar = 'bar';

console.log(obj); // { bar: 'bar' }
console.log(sym in obj); // true
console.log(obj[sym]); // foo
console.log(Object.keys(obj)); // ['bar']
```

注意，symbols 键不会被在 Object.keys() 返回。这也是为了满足向后兼容性。旧版本的 JavaScript 没有 symbol 数据类型，因此不应该从旧的 Object.keys() 方法中被返回。

乍一看，这就像是可以用 symbols 在对象上创建私有属性！许多其他编程语言可以在其类中有私有属性，而 JavaScript 却遗漏了这种功能，长期以来被视为其语法的一种缺点。

不幸的是，与该对象交互的代码仍然可以访问对象那些键为 symbols 的属性。甚至是在调用代码自己**无法**访问 symbol 的情况下也有可能发生。 例如，Reflect.ownKeys() 方法能够得到一个对象的**所有**键的列表，包括字符串和 symbols：

```
function tryToAddPrivate(obj) {
  obj[Symbol('Pseudo Private')] = 42;
}

const obj = { prop: 'hello' };
tryToAddPrivate(obj);

console.log(Reflect.ownKeys(obj));

console.log(obj[Reflect.ownKeys(obj)[1]]); // 42
```

防止属性名冲突

Symbol 类型可能会对获取 JavaScript 中对象的私有属性不利。它们之所以有用的另一个理由是，当不同的库希望向对象添加属性时 symbols 可以避免命名冲突的风险。

如果有两个不同的库希望将某种元数据附加到一个对象上，两者可能都想在对象上设置某种标识符。仅仅使用两个字符串类型的 id 作为键来标识，多个库使用相同键的风险就会很高。

```
function lib1tag(obj) {
  obj.id = 42;
}

function lib2tag(obj) {
  obj.id = 369;
}
```

应用 symbols，每个库都可以通过实例化 Symbol 类生成所需的 symbols。然后不管什么时候，都可以在相应的对象上检查、赋值 symbols 对应的键值。

```
const library1property = Symbol('lib1');
function lib1tag(obj) {
  obj[library1property] = 42;
}

const library2property = Symbol('lib2');
function lib2tag(obj) {
  obj[library2property] = 369;
}
```

基于这个原因 symbols **确实**有益于 JavaScript。

然而，你可能会怀疑，为什么每个库不能在实例化时简单地生成一个随机字符串，或者使用一个特殊的命名空间？

```
const library1property = uuid(); // 随机方法
function lib1tag(obj) {
  obj[library1property] = 42;
}

const library2property = 'LIB2-NAMESPACE-id'; // namespaced approach
function lib2tag(obj) {
  obj[library2property] = 369;
}
```

你有可能是正确的，上面的两种方法与使用 symbols 的方法很相似。除非两个库使用了相同的属性名，否则不会有冲突的风险。

在这一点上，机灵的读者会指出，这两种方法并不完全相同。具有唯一名称的属性名仍然有一个缺点：它们的键非常容易找到，特别是当运行代码来迭代键或以其他方式序列化对象时。请考虑以下示例：

```
const library2property = 'LIB2-NAMESPACE-id'; // namespaced
function lib2tag(obj) {
  obj[library2property] = 369;
}

const user = {
  name: 'Thomas Hunter II',
  age: 32
};

lib2tag(user);
```

```
JSON.stringify(user);
// '{"name":"Thomas Hunter II","age":32,"LIB2-NAMESPACE-id":369}'
```

如果我们为对象的属性名使用了一个 symbol，那么 JSON 的输出将不包含 symbol 对应的值。为什么会这样？因为仅仅是 JavaScript 支持了 symbols，并不意味着 JSON 规范也改变了！JSON 只允许字符串作为键，而 JavaScript 不会尝试在最终的 JSON 负载中呈现 symbol 属性。

我们可以通过使用 object.defineproperty()，轻松纠正库对象字符串污染 JSON 输出的问题：

```
const library2property = uuid(); // namespaced approach
function lib2tag(obj) {
  Object.defineProperty(obj, library2property, {
    enumerable: false,
    value: 369
  });
}

const user = {
  name: 'Thomas Hunter II',
  age: 32
};

lib2tag(user);
// '{"name":"Thomas Hunter II","age":32,"f468c902-26ed-4b2e-81d6-
5775ae7eec5d":369}
console.log(JSON.stringify(user));
console.log(user[library2property]); // 369
```

通过将字符串键的可枚举描述符设置为 false 来“隐藏”字符串键的行为非常类似于 symbol 键。它们通过 Object.keys() 遍历也看不到，但可以通过 Reflect.ownKeys()显示，如下所示：

```
const obj = {};
obj[Symbol()] = 1;
Object.defineProperty(obj, 'foo', {
  enumberable: false,
  value: 2
});

console.log(Object.keys(obj)); // []
```

```
console.log(Reflect.ownKeys(obj)); // [ 'foo', Symbol() ]
console.log(JSON.stringify(obj)); // {}
```

在这一点上，我们**几乎**重新创建了 symbols。隐藏的字符串属性和 symbols 都对序列化程序隐身。这两种属性都可以使用 Reflect.ownKeys()方法提取，因此实际上并不是私有的。假设我们对字符串属性使用某种命名空间/随机值，那么我们就消除了多个库意外发生命名冲突的风险。

但是，仍然有一个微小的差异。由于字符串是不可变的，Symbol 始终保证是唯一的，因此仍有可能生成相同的字符串并产生冲突。从数学角度来说，意味着 symbols 确实提供了我们无法从字符串中获得的好处。

在 Node.js 中，检查对象时(例如使用 console.log())，如果遇到对象上名为 inspect 的方法，则调用该函数，并将输出表示成对象的日志。可以想象，这种行为并不是每个人都期望的，通常命名为 inspect 的方法经常与用户创建的对象发生冲突。现在有 symbol 可用来实现这个功能，并且可以在 require('util').inspection.custom 中使用。inspect 方法在 Node.js v10 中被废弃，在 v11 中完全被忽略。现在没有人会因为意外改变 inspect 的行为!