7.3.7 Symbol.unscopables

符号 @@unscopables 可以被定义为任意对象的对象属性（Symbol.unscopables），用来指示使用 with 语句时哪些属性可以或不可以暴露为词法变量。

考虑：

```
var o = { a:1, b:2, c:3 },
    a = 10, b = 20, c = 30;

o[Symbol.unscopables] = {
    a: false,
    b: true,
    c: false
};

with (o) {
    console.log( a, b, c );      // 1 20 3
}
```

@@unscopables 对象中的 true 表示这个属性应该是 unscopable 的，因此会从词法作用域变量中被过滤出去。false 表示可以将其包含到词法作用域变量中。

strict 模式下不允许 with 语句，因此应当被认为是语言的过时特性。不要使用它。参见本系列《你不知道的 JavaScript（上卷）》第一部分获取更多信息。因为应该避免使用 with，所以符号 @@unscopables 也没有太大意义。

7.4 代理

ES6 中新增的最明显的元编程特性之一是 Proxy（代理）特性。

代理是一种由你创建的特殊的对象，它“封装”另一个普通对象——或者说挡在这个普通对象的前面。你可以在代理对象上注册特殊的处理函数（也就是 trap），代理上执行各种操作的时候会调用这个程序。这些处理函数除了把操作转发给原始目标 / 被封装对象之外，还有机会执行额外的逻辑。

你可以在代理上定义的 trap 处理函数的一个例子是 get，当你试图访问对象属性的时候，它拦截 [[Get]] 运算。考虑：

```
var obj = { a: 1 },
    handlers = {
        get(target,key,context) {
            // 注意：target === obj,
            // context === pobj
            console.log( "accessing: ", key );
            return Reflect.get(
```

```
                target, key, context
            );
        }
    },
    pobj = new Proxy( obj, handlers );

obj.a;
// 1

pobj.a;
// accessing: a
// 1
```

我们在 handlers（`Proxy(..)` 的第二个参数）对象上声明了一个 `get(..)` 处理函数命名方法，它接受一个 target 对象的引用（`obj`）、key 属性名 (`"a"`) **粗体文字**以及 `self`/ 接收者 / 代理（`pobj`）。

在跟踪语句 `console.log(..)` 之后，我们把对 `obj` 的操作通过 `Reflect.get(..)` “转发”。下一小节中会介绍 API`Reflect`，这里只要了解每个可用的代理 trap 都有一个对应的同名 `Reflect` 函数即可。

这里的映射是有意对称的。每个代理处理函数在对应的元编程任务执行的时候进行拦截，而每个 `Reflect` 工具在一个对象上执行相应的元编程任务。每个代理处理函数都有一个自动调用相应的 `Reflect` 工具的默认定义。几乎可以确定 `Proxy` 和 `Reflect` 总是这么协同工作的。

下面所列出的是在目标对象 / 函数代理上可以定义的处理函数，以及它们如何 / 何时被触发。

`get(..)`
: 通过 `[[Get]]`，在代理上访问一个属性（`Reflect.get(..)`、`.` 属性运算符或 `[ .. ]` 属性运算符）

`set(..)`
: 通过 `[[Set]]`，在代理上设置一个属性值（`Reflect.set(..)`、赋值运算符 `=` 或目标为对象属性的解构赋值）

`deleteProperty(..)`
: 通过 `[[Delete]]`，从代理对象上删除一个属性（`Reflect.deleteProperty(..)` 或 `delete`）

`apply(..)`（**如果目标为函数**）
: 通过 `[[Call]]`，将代理作为普通函数 / 方法调用（`Reflect.apply(..)`、`call(..)`、`apply(..)` 或 `(..)` 调用运算符）

construct(..)（**如果目标为构造函数**）

通过 [[Construct]]，将代理作为构造函数调用（Reflect.construct(..) 或 new）

getOwnPropertyDescriptor(..)

通过 [[GetOwnProperty]]，从代理中提取一个属性描述符（Object.getOwnPropertyDescriptor(..) 或 Reflect.getOwnPropertyDescriptor(..)）

defineProperty(..)

通过 [[DefineOwnProperty]]，在代理上设置一个属性描述符（Object.defineProperty(..) 或 Reflect.defineProperty(..)）

getPrototypeOf(..)

通过 [[GetPrototypeOf]]，得到代理的 [[Prototype]]（Object.getPrototypeOf(..)、Reflect.getPrototypeOf(..)、__proto__、Object#isPrototypeOf(..) 或 instanceof）

setPrototypeOf(..)

通过 [[SetPrototypeOf]]，设置代理的 [[Prototype]]（Object.setPrototypeOf(..)、Reflect.setPrototypeOf(..) 或 __proto__）

preventExtensions(..)

通过 [[PreventExtensions]]，使得代理变成不可扩展的（Object.prevent Extensions(..) 或 Reflect.preventExtensions(..)）

isExtensible(..)

通过 [[IsExtensible]]，检测代理是否可扩展（Object.isExtensible(..) 或 Reflect.isExtensible(..)）

ownKeys(..)

通过 [[OwnPropertyKeys]]，提取代理自己的属性和 / 或符号属性（Object.keys(..)、Object.getOwnPropertyNames(..)、Object.getOwnSymbolProperties(..)、Reflect.ownKeys(..) 或 JSON.stringify(..)）

enumerate(..)

通过 [[Enumerate]]，取得代理拥有的和“继承来的”可枚举属性的迭代器（Reflect.enumerate(..) 或 for..in）

has(..)

通过 [[HasProperty]]，检查代理是否拥有或者“继承了”某个属性（Reflect.has(..)、Object#hasOwnProperty(..) 或 "prop" in obj）

要了解这里每个元编程任务的更多信息，参见 7.5 节。

除了上面列出的会触发各种 trap 的动作，某些 trap 是由其他 trap 的默认动作间接触发的。比如：

```
var handlers = {
        getOwnPropertyDescriptor(target,prop) {
            console.log(
                "getOwnPropertyDescriptor"
            );
            return Object.getOwnPropertyDescriptor(
                target, prop
            );
        },
        defineProperty(target,prop,desc){
            console.log( "defineProperty" );
            return Object.defineProperty(
                target, prop, desc
            );
        }
    },
    proxy = new Proxy( {}, handlers );

proxy.a = 2;
// getOwnPropertyDescriptor
// defineProperty
```

`getOwnPropertyDescriptor(..)` 和 `defineProperty(..)` 处理函数是在设定属性值（不管是新增的还是更新已有的）时由默认 `set(..)` 处理函数的步骤触发的。如果你也自定义了 `set(..)` 处理函数，那么在 `context`（不是 `target`！）上可以（也可以不）进行相应的调用，这些调用会触发这些代理 trap。

7.4.1 代理局限性

可以在对象上执行的很广泛的一组基本操作都可以通过这些元编程处理函数 trap。但有一些操作是无法（至少现在）拦截的。

比如，下面这些操作都不会 trap 并从代理 `pobj` 转发到目标 `obj`：

```
var obj = { a:1, b:2 },
    handlers = { .. },
    pobj = new Proxy( obj, handlers );

typeof obj;
String( obj );
```

```
obj + "";
obj == pobj;
obj === pobj
```

也许在未来，语言中的这些底层基本运算可能会有更多变成可拦截的，那将为我们从内部扩展 JavaScript 提供更强大的能力。

代理处理函数总会有一些不变性（invariant），亦即不能被覆盖的行为。比如，`isExtensible(..)` 处理函数的返回值总会被类型转换为 `boolean`。这些不变性限制了自定义代理行为的能力，但它们的目的只是为了防止你创建诡异或罕见（或者不一致）的行为。这些不变性条件非常复杂，所以这里我们不会完整介绍，这篇文章（http://www.2ality.com/2014/12/es6-proxies.html#invariants）对此给出了很棒的介绍。

7.4.2 可取消代理

普通代理总是陷入到目标对象，并且在创建之后不能修改——只要还保持着对这个代理的引用，代理的机制就将维持下去。但是，可能会存在这样的情况，比如你想要创建一个在你想要停止它作为代理时便可以被停用的代理。解决方案是创建**可取消代理**（revocable proxy）：

```
var obj = { a: 1 },
    handlers = {
        get(target,key,context) {
            // 注意：target === obj,
            // context === pobj
            console.log( "accessing: ", key );
            return target[key];
        }
    },
    { proxy: pobj, revoke: prevoke } =
        Proxy.revocable( obj, handlers );

pobj.a;
// accessing: a
// 1

// 然后：
prevoke();

pobj.a;
// TypeError
```

可取消代理用 `Proxy.revocable(..)` 创建，这是一个普通函数，而不像 `Proxy(..)` 一样是构造器。除此之外，它接收同样的两个参数：target 和 handlers。

和 `new Proxy(..)` 不一样，`Proxy.revocable(..)` 的返回值不是代理本身。而是一个有两个属性——proxy 和 revode 的对象，我们使用对象解构（参见 2.4 节）把这两个属性分别赋

给变量 pobj 和 prevoke()。

一旦可取消代理被取消，任何对它的访问（触发它的任意 trap）都会抛出 TypeError。

可取消代理的一个可能应用场景是，在你的应用中把代理分发到第三方，其中管理你的模型数据，而不是给出真实模型本身的引用。如果你的模型对象改变或者被替换，就可以使分发出去的代理失效，这样第三方能够（通过错误！）知晓变化并请求更新到这个模型的引用。

7.4.3 使用代理

这些处理函数为元编程带来的好处是显而易见的。我们可以拦截（并覆盖）对象的几乎所有行为，这意味着我们可以以强有力的方式扩展对象特性超出核心 JavaScript 内容。这里将会通过几种模式实例来探索这些可能性。

1. 代理在先，代理在后

我们在前面介绍过，通常可以把代理看作是对目标对象的“包装”。在这种意义上，代理成为了代码交互的主要对象，而实际目标对象保持隐藏 / 被保护的状态。

你可能这么做是因为你想要把对象传入到某个无法被完全“信任”的环境，因此需要为对它的访问增强规范性，而不是把对象本身传入。

考虑：

```
var messages = [],
    handlers = {
        get(target,key) {
            // 字符串值?
            if (typeof target[key] == "string") {
                // 过滤掉标点符号
                return target[key]
                    .replace( /[^\w]/g, "" );
            }

            // 所有其他的传递下去
            return target[key];
        },
        set(target,key,val) {
            // 设定唯一字符串，改为小写
            if (typeof val == "string") {
                val = val.toLowerCase();
                if (target.indexOf( val ) == -1) {
                    target.push(
                        val.toLowerCase()
                    );
                }
            }
            return true;
```

```
        }
    },
    messages_proxy =
        new Proxy( messages, handlers );

// 其他某处：
messages_proxy.push(
    "heLLo...", 42, "wOrlD!!", "WoRld!!"
);

messages_proxy.forEach( function(val){
    console.log(val);
} );
// hello world

messages.forEach( function(val){
    console.log(val);
} );
// hello... world!!
```

我称之为**代理在先**（proxy first）设计，因为我们首先（主要、完全）与代理交互。

通过与 `messages_proxy` 交互来增加某些特殊的规则，这些是 `messages` 本身没有的。我们只在值为字符串并且是唯一值的时候才添加这个元素；我们还将这个值变为小写。在从 `messages_proxy` 提取值的时候，我们过滤掉了字符串中的所有标点符号。

另外，我们也可以完全反转这个模式，让目标与代理交流，而不是代理与目标交流。这样，代码只能与主对象交互。这个回退方式的最简单实现就是把 proxy 对象放到主对象的 `[[Prototype]]` 链中。

考虑：

```
var handlers = {
        get(target,key,context) {
            return function() {
                context.speak(key + "!");
            };
        }
    },
    catchall = new Proxy( {}, handlers ),
    greeter = {
        speak(who = "someone") {
            console.log( "hello", who );
        }
    };

// 设定greeter回退到catchall
Object.setPrototypeOf( greeter, catchall );

greeter.speak();                    // hello someone
greeter.speak( "world" );           // hello world

greeter.everyone();                 // hello everyone!
```

这里直接与 greeter 而不是 catchall 交流。当我们调用 speak(..) 的时候，它在 greeter 上被找到并直接使用。但是当我们试图访问像 everyone() 这样的方法的时候，这个函数在 greeter 上并不存在。

默认的对象属性行为是检查 [[Prototype]] 链（参见本系列《你不知道的 JavaScript（上卷）》第二部分），所以会查看 catchall 是否有 everyone 属性。然后代理的 get() 处理函数介入并返回一个用访问的属性名（"everyone"）调用 speak(..) 的函数。

我把这个模式称为**代理在后**（proxy last），因为在这里代理只作为最后的保障。

2. "No Such Property/Method"

有一个关于 JavaScript 的常见抱怨，在你试着访问或设置一个还不存在的属性时，默认情况下对象不是非常具有防御性。你可能希望预先定义好一个对象的所有属性 / 方法之后，访问不存在的属性名时能够抛出一个错误。

我们可以通过代理实现这一点，代理在先或代理在后设计都可以。两种情况我们都考虑一下：

```
var obj = {
        a: 1,
        foo() {
            console.log( "a:", this.a );
        }
    },
    handlers = {
        get(target,key,context) {
            if (Reflect.has( target, key )) {
                return Reflect.get(
                    target, key, context
                );
            }
            else {
                throw "No such property/method!";
            }
        },
        set(target,key,val,context) {
            if (Reflect.has( target, key )) {
                return Reflect.set(
                    target, key, val, context
                );
            }
            else {
                throw "No such property/method!";
            }
        }
    },
    pobj = new Proxy( obj, handlers );

pobj.a = 3;
pobj.foo();      // a: 3
```

```
pobj.b = 4;      // Error: No such property/method!
pobj.bar();      // Error: No such property/method!
```

对于 get(..) 和 set(..)，我们都只在目标对象的属性存在的时候才转发这个操作；否则抛出错误。主对象代码应该与代理对象（pobj）交流，因为它截获这些动作以提供保护。

现在，考虑转换为代理在后设计：

```
var handlers = {
        get() {
            throw "No such property/method!";
        },
        set() {
            throw "No such property/method!";
        }
    },
    pobj = new Proxy( {}, handlers ),
    obj = {
        a: 1,
        foo() {
            console.log( "a:", this.a );
        }
    };

// 设定obj回退到pobj
Object.setPrototypeOf( obj, pobj );

obj.a = 3;
obj.foo();        // a: 3

obj.b = 4;        // Error: No such property/method!
obj.bar();        // Error: No such property/method!
```

考虑到处理函数的定义方式，这里的代理在后设计更简单一些。与截获 [[Get]] 和 [[Set]] 操作并且只在目标属性存在情况下才转发不同，我们依赖于这样一个事实：如果 [[Get]] 或 [[Set]] 进入我们的 pobj 回退，此时这个动作已经遍历了整个 [[Prototype]] 链并且没有发现匹配的属性。这时候我们可以自由抛出错误。不错吧?

3. 代理 hack [[Prototype]] 链

[[Prototype]] 机制运作的主要通道是 [[Get]] 运算。当直接对象中没有找到一个属性的时候，[[Get]] 会自动把这个运算转给 [[Prototype]] 对象处理。

这意味着你可以使用代理的 get(..) trap 来模拟或扩展这个 [[Prototype]] 机制的概念。

我们将考虑的第一个 hack 就是创建两个对象，通过 [[Prototype]] 连成环状（或者，至少看起来是这样！）。实际上并不能创建一个真正的 [[Prototype]] 环，因为引擎会抛出错误。但是可以用代理模拟!

考虑：

```
var handlers = {
        get(target,key,context) {
            if (Reflect.has( target, key )) {
                return Reflect.get(
                    target, key, context
                );
            }
            // 伪环状[[Prototype]]
            else {
                return Reflect.get(
                    target[
                        Symbol.for( "[[Prototype]]" )
                    ],
                    key,
                    context
                );
            }
        }
    },
    obj1 = new Proxy(
        {
            name: "obj-1",
            foo() {
                console.log( "foo:", this.name );
            }
        },
        handlers ),
    },
    obj2 = Object.assign(
        Object.create( obj1 ),
        {
            name: "obj-2",
            bar() {
                console.log( "bar:", this.name );
                this.foo();
            }
        }
    );

// 伪环状[[Prototype]]链接
obj1[ Symbol.for( "[[Prototype]]" ) ] = obj2;

obj1.bar();
// bar: obj-1 <-- 通过代理伪装[[Prototype]]
// foo: obj-1 <-- this上下文依然保留着

obj2.foo();
// foo: obj-2 <-- 通过[[Prototype]]
```

在这个例子中，我们不需要代理 / 转发 [[Set]]，所以比较简单。要完整模拟 [[Prototype]]，需要实现一个 set(..) 处理函数来搜索 [[Prototype]] 链寻找匹配的属性，并遵守其描述符特性（比如 set，可写的）。参见本系列《你不知道的 JavaScript（上卷）》第二部分。