```
var myObject = {
    a: 2
};

myObject.a; // 2
```

myObject.a 是一次属性访问，但是这条语句并不仅仅是在 myObjet 中查找名字为 a 的属性，虽然看起来好像是这样。

在语言规范中，myObject.a 在 myObject 上实际上是实现了 [[Get]] 操作（有点像函数调用：[[Get]]()）。对象默认的内置 [[Get]] 操作首先在对象中查找是否有名称相同的属性，如果找到就会返回这个属性的值。

然而，如果没有找到名称相同的属性，按照 [[Get]] 算法的定义会执行另外一种非常重要的行为。我们会在第 5 章中介绍这个行为（其实就是遍历可能存在的 [[Prototype]] 链，也就是原型链）。

如果无论如何都没有找到名称相同的属性，那 [[Get]] 操作会返回值 undefined：

```
var myObject = {
    a:2
};

myObject.b; // undefined
```

注意，这种方法和访问变量时是不一样的。如果你引用了一个当前词法作用域中不存在的变量，并不会像对象属性一样返回 undefined，而是会抛出一个 ReferenceError 异常：

```
var myObject = {
    a: undefined
};

myObject.a; // undefined

myObject.b; // undefined
```

从返回值的角度来说，这两个引用没有区别——它们都返回了 undefined。然而，尽管乍看之下没什么区别，实际上底层的 [[Get]] 操作对 myObject.b 进行了更复杂的处理。

由于仅根据返回值无法判断出到底变量的值为 undefined 还是变量不存在，所以 [[Get]] 操作返回了 undefined。不过稍后我们会介绍如何区分这两种情况。

3.3.8 [[Put]]

既然有可以获取属性值的 [[Get]] 操作，就一定有对应的 [[Put]] 操作。

你可能会认为给对象的属性赋值会触发 [[Put]] 来设置或者创建这个属性。但是实际情况

并不完全是这样。

[[Put]] 被触发时，实际的行为取决于许多因素，包括对象中是否已经存在这个属性（这是最重要的因素）。

如果已经存在这个属性，[[Put]] 算法大致会检查下面这些内容。

1. 属性是否是访问描述符（参见 3.3.9 节）？如果是并且存在 setter 就调用 setter。
2. 属性的数据描述符中 writable 是否是 false？如果是，在非严格模式下静默失败，在严格模式下抛出 TypeError 异常。
3. 如果都不是，将该值设置为属性的值。

如果对象中不存在这个属性，[[Put]] 操作会更加复杂。我们会在第 5 章讨论 [[Prototype]] 时详细进行介绍。

3.3.9 Getter和Setter

对象默认的 [[Put]] 和 [[Get]] 操作分别可以控制属性值的设置和获取。

在语言的未来 / 高级特性中，有可能可以改写整个对象（不仅仅是某个属性）的默认 [[Get]] 和 [[Put]] 操作。这已经超出了本书的讨论范围，但是将来“你不知道的 JavaScript”系列丛书中有可能会对这个问题进行探讨。

在 ES5 中可以使用 getter 和 setter 部分改写默认操作，但是只能应用在单个属性上，无法应用在整个对象上。getter 是一个隐藏函数，会在获取属性值时调用。setter 也是一个隐藏函数，会在设置属性值时调用。

当你给一个属性定义 getter、setter 或者两者都有时，这个属性会被定义为“访问描述符”（和“数据描述符”相对）。对于访问描述符来说，JavaScript 会忽略它们的 value 和 writable 特性，取而代之的是关心 set 和 get（还有 configurable 和 enumerable）特性。

思考下面的代码：

```
var myObject = {
    // 给 a 定义一个 getter
    get a() {
        return 2;
    }
};

Object.defineProperty(
    myObject,  // 目标对象
    "b",         // 属性名
```

```
    {           // 描述符
        // 给 b 设置一个 getter
        get: function(){ return this.a * 2 },

        // 确保 b 会出现在对象的属性列表中
        enumerable: true
    }
);

myObject.a; // 2

myObject.b; // 4
```

不管是对象文字语法中的 `get a() { .. }`，还是 `defineProperty(..)` 中的显式定义，二者都会在对象中创建一个不包含值的属性，对于这个属性的访问会自动调用一个隐藏函数，它的返回值会被当作属性访问的返回值：

```
var myObject = {
    // 给 a 定义一个 getter
    get a() {
        return 2;
    }
};

myObject.a = 3;

myObject.a; // 2
```

由于我们只定义了 a 的 getter，所以对 a 的值进行设置时 `set` 操作会忽略赋值操作，不会抛出错误。而且即便有合法的 setter，由于我们自定义的 getter 只会返回 2，所以 `set` 操作是没有意义的。

为了让属性更合理，还应当定义 setter，和你期望的一样，setter 会覆盖单个属性默认的 `[[Put]]`（也被称为赋值）操作。通常来说 getter 和 setter 是成对出现的（只定义一个的话通常会产生意料之外的行为）：

```
var myObject = {
    // 给 a 定义一个 getter
    get a() {
        return this._a_;
    },

    // 给 a 定义一个 setter
    set a(val) {
        this._a_ = val * 2;
    }
};

myObject.a = 2;

myObject.a; // 4
```

在本例中，实际上我们把赋值（[[Put]]）操作中的值 2 存储到了另一个变量 _a_ 中。名称 _a_ 只是一种惯例，没有任何特殊的行为——和其他普通属性一样。

3.3.10 存在性

前面我们介绍过，如 myObject.a 的属性访问返回值可能是 undefined，但是这个值有可能是属性中存储的 undefined，也可能是因为属性不存在所以返回 undefined。那么如何区分这两种情况呢？

我们可以在不访问属性值的情况下判断对象中是否存在这个属性：

```
var myObject = {
    a:2
};

("a" in myObject); // true
("b" in myObject); // false

myObject.hasOwnProperty( "a" ); // true
myObject.hasOwnProperty( "b" ); // false
```

in 操作符会检查属性是否在对象及其 [[Prototype]] 原型链中（参见第 5 章）。相比之下，hasOwnProperty(..) 只会检查属性是否在 myObject 对象中，不会检查 [[Prototype]] 链。在第 5 章讲解 [[Prototype]] 时我们会详细介绍这两者的区别。

所有的普通对象都可以通过对于 Object.prototype 的委托（参见第 5 章）来访问 hasOwnProperty(..)，但是有的对象可能没有连接到 Object.prototype（通过 Object.create(null) 来创建——参见第 5 章）。在这种情况下，形如 myObejct.hasOwnProperty(..) 就会失败。

这时可以使用一种更加强硬的方法来进行判断：Object.prototype.hasOwnProperty.call(myObject,"a")，它借用基础的 hasOwnProperty(..) 方法并把它显式绑定（参见第 2 章）到 myObject 上。

看起来 in 操作符可以检查容器内是否有某个值，但是它实际上检查的是某个属性名是否存在。对于数组来说这个区别非常重要，4 in [2, 4, 6] 的结果并不是你期待的 True，因为 [2, 4, 6] 这个数组中包含的属性名是 0、1、2，没有 4。

1. 枚举

之前介绍 enumerable 属性描述符特性时我们简单解释过什么是“可枚举性”，现在详细介

绍一下：

```
var myObject = { };

Object.defineProperty(
    myObject,
    "a",
    // 让 a 像普通属性一样可以枚举
    { enumerable: true, value: 2 }
);

Object.defineProperty(
    myObject,
    "b",
    // 让 b 不可枚举
    { enumerable: false, value: 3 }
);

myObject.b; // 3
("b" in myObject); // true
myObject.hasOwnProperty( "b" ); // true

// .......

for (var k in myObject) {
    console.log( k, myObject[k] );
}
// "a" 2
```

可以看到，myObject.b 确实存在并且有访问值，但是却不会出现在 for..in 循环中（尽管可以通过 in 操作符来判断是否存在）。原因是“可枚举”就相当于“可以出现在对象属性的遍历中”。

在数组上应用 for..in 循环有时会产生出人意料的结果，因为这种枚举不仅会包含所有数值索引，还会包含所有可枚举属性。最好只在对象上应用 for..in 循环，如果要遍历数组就使用传统的 for 循环来遍历数值索引。

也可以通过另一种方式来区分属性是否可枚举：

```
var myObject = { };

Object.defineProperty(
    myObject,
    "a",
    // 让 a 像普通属性一样可以枚举
    { enumerable: true, value: 2 }
);

Object.defineProperty(
```

```
    myObject,
    "b",
    // 让 b 不可枚举
    { enumerable: false, value: 3 }
);

myObject.propertyIsEnumerable( "a" ); // true
myObject.propertyIsEnumerable( "b" ); // false

Object.keys( myObject ); // ["a"]
Object.getOwnPropertyNames( myObject ); // ["a", "b"]
```

propertyIsEnumerable(..) 会检查给定的属性名是否直接存在于对象中（而不是在原型链上）并且满足 enumerable:true。

Object.keys(..) 会返回一个数组，包含所有可枚举属性，Object.getOwnPropertyNames(..) 会返回一个数组，包含所有属性，无论它们是否可枚举。

in 和 hasOwnProperty(..) 的区别在于是否查找 [[Prototype]] 链，然而，Object.keys(..) 和 Object.getOwnPropertyNames(..) 都只会查找对象直接包含的属性。

（目前）并没有内置的方法可以获取 in 操作符使用的属性列表（对象本身的属性以及 [[Prototype]] 链中的所有属性，参见第 5 章）。不过你可以递归遍历某个对象的整条 [[Prototype]] 链并保存每一层中使用 Object.keys(..) 得到的属性列表——只包含可枚举属性。

3.4 遍历

for..in 循环可以用来遍历对象的可枚举属性列表（包括 [[Prototype]] 链）。但是如何遍历属性的值呢？

对于数值索引的数组来说，可以使用标准的 for 循环来遍历值：

```
var myArray = [1, 2, 3];

for (var i = 0; i < myArray.length; i++) {
    console.log( myArray[i] );
}
// 1 2 3
```

这实际上并不是在遍历值，而是遍历下标来指向值，如 myArray[i]。

ES5 中增加了一些数组的辅助迭代器，包括 forEach(..)、every(..) 和 some(..)。每种辅助迭代器都可以接受一个回调函数并把它应用到数组的每个元素上，唯一的区别就是它们对于回调函数返回值的处理方式不同。

forEach(..) 会遍历数组中的所有值并忽略回调函数的返回值。every(..) 会一直运行直到

回调函数返回 false（或者“假”值），some(..) 会一直运行直到回调函数返回 true（或者“真”值）。

every(..) 和 some(..) 中特殊的返回值和普通 for 循环中的 break 语句类似，它们会提前终止遍历。

使用 for..in 遍历对象是无法直接获取属性值的，因为它实际上遍历的是对象中的所有可枚举属性，你需要手动获取属性值。

遍历数组下标时采用的是数字顺序（for 循环或者其他迭代器），但是遍历对象属性时的顺序是不确定的，在不同的 JavaScript 引擎中可能不一样。因此，在不同的环境中需要保证一致性时，一定不要相信任何观察到的顺序，它们是不可靠的。

那么如何直接遍历值而不是数组下标（或者对象属性）呢？幸好，ES6 增加了一种用来遍历数组的 for..of 循环语法（如果对象本身定义了迭代器的话也可以遍历对象）：

```
var myArray = [ 1, 2, 3 ];

for (var v of myArray) {
    console.log( v );
}
// 1
// 2
// 3
```

for..of 循环首先会向被访问对象请求一个迭代器对象，然后通过调用迭代器对象的 next() 方法来遍历所有返回值。

数组有内置的 @@iterator，因此 for..of 可以直接应用在数组上。我们使用内置的 @@iterator 来手动遍历数组，看看它是怎么工作的：

```
var myArray = [ 1, 2, 3 ];
var it = myArray[Symbol.iterator]();

it.next(); // { value:1, done:false }
it.next(); // { value:2, done:false }
it.next(); // { value:3, done:false }
it.next(); // { done:true }
```

我们使用 ES6 中的符号 Symbol.iterator 来获取对象的 @@iterator 内部属性。之前我们简单介绍过符号（Symbol，参见 3.3.1 节），跟这里的原理是相同的。引用类似 iterator 的特殊属性时要使用符号名，而不是符号包含的值。此外，虽然看起来很像一个对象，但是 @@iterator 本身并不是一个迭代器对象，而是一个返回迭代器对象的函数——这点非常精妙并且非常重要。

如你所见，调用迭代器的 next() 方法会返回形式为 { value: .. , done: .. } 的值，value 是当前的遍历值，done 是一个布尔值，表示是否还有可以遍历的值。

注意，和值“3”一起返回的是 done:false，乍一看好像很奇怪，你必须再调用一次 next() 才能得到 done:true，从而确定完成遍历。这个机制和 ES6 中发生器函数的语义相关，不过已经超出了我们的讨论范围。

和数组不同，普通的对象没有内置的 @@iterator，所以无法自动完成 for..of 遍历。之所以要这样做，有许多非常复杂的原因，不过简单来说，这样做是为了避免影响未来的对象类型。

当然，你可以给任何想遍历的对象定义 @@iterator，举例来说：

```
var myObject = {
    a: 2,
    b: 3
};

Object.defineProperty( myObject, Symbol.iterator, {
    enumerable: false,
    writable: false,
    configurable: true,
    value: function() {
        var o = this;
        var idx = 0;
        var ks = Object.keys( o );
        return {
            next: function() {
                return {
                    value: o[ks[idx++]],
                    done: (idx > ks.length)
                };
            }
        };
    }
} );

// 手动遍历 myObject
var it = myObject[Symbol.iterator]();
it.next(); // { value:2, done:false }
it.next(); // { value:3, done:false }
it.next(); // { value:undefined, done:true }

// 用 for..of 遍历 myObject
for (var v of myObject) {
    console.log( v );
}
// 2
// 3
```

我们使用 Object.defineProperty(..) 定义了我们自己的 @@iterator（主要是为了让它不可枚举），不过注意，我们把符号当作可计算属性名（本章之前有介绍）。此外，也可以直接在定义对象时进行声明，比如 var myObject = { a:2, b:3, [Symbol.iterator]: function() { /* .. */ } }。

for..of 循环每次调用 myObject 迭代器对象的 next() 方法时，内部的指针都会向前移动并返回对象属性列表的下一个值（再次提醒，需要注意遍历对象属性 / 值时的顺序）。

代码中的遍历非常简单，只是传递了属性本身的值。不过只要你愿意，当然也可以在自定义的数据结构上实现各种复杂的遍历。对于用户定义的对象来说，结合 for..of 循环和自定义迭代器可以组成非常强大的对象操作工具。

比如说，一个 Pixel 对象（有 x 和 y 坐标值）列表可以按照距离原点的直线距离来决定遍历顺序，也可以过滤掉“太远”的点，等等。只要迭代器的 next() 调用会返回 { value: .. } 和 { done: true }，ES6 中的 for..of 就可以遍历它。

实际上，你甚至可以定义一个“无限”迭代器，它永远不会“结束”并且总会返回一个新值（比如随机数、递增值、唯一标识符，等等）。你可能永远不会在 for..of 循环中使用这样的迭代器，因为它永远不会结束，你的程序会被挂起：

```
var randoms = {
    [Symbol.iterator]: function() {
        return {
            next: function() {
                return { value: Math.random() };
            }
        };
    }
};

var randoms_pool = [];
for (var n of randoms) {
    randoms_pool.push( n );

    // 防止无限运行!
    if (randoms_pool.length === 100) break;
}
```

这个迭代器会生成“无限个”随机数，因此我们添加了一条 break 语句，防止程序被挂起。

3.5 小结

JavaScript 中的对象有字面形式（比如 var a = { .. }）和构造形式（比如 var a = new Array(..)）。字面形式更常用，不过有时候构造形式可以提供更多选项。

许多人都以为“JavaScript 中万物都是对象”，这是错误的。对象是 6 个（或者是 7 个，取决于你的观点）基础类型之一。对象有包括 `function` 在内的子类型，不同子类型具有不同的行为，比如内部标签 `[object Array]` 表示这是对象的子类型数组。

对象就是键 / 值对的集合。可以通过 `.propName` 或者 `["propName"]` 语法来获取属性值。访问属性时，引擎实际上会调用内部的默认 `[[Get]]` 操作（在设置属性值时是 `[[Put]]`），`[[Get]]` 操作会检查对象本身是否包含这个属性，如果没找到的话还会查找 `[[Prototype]]` 链（参见第 5 章）。

属性的特性可以通过属性描述符来控制，比如 `writable` 和 `configurable`。此外，可以使用 `Object.preventExtensions(..)`、`Object.seal(..)` 和 `Object.freeze(..)` 来设置对象（及其属性）的不可变性级别。

属性不一定包含值——它们可能是具备 getter/setter 的“访问描述符”。此外，属性可以是可枚举或者不可枚举的，这决定了它们是否会出现在 `for..in` 循环中。

你可以使用 ES6 的 `for..of` 语法来遍历数据结构（数组、对象，等等）中的值，`for..of` 会寻找内置或者自定义的 `@@iterator` 对象并调用它的 `next()` 方法来遍历数据值。