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

var idx;

if (wantA) {
    idx = "a";
}

// 之后

console.log( myObject[idx] ); // 2
```

在对象中，属性名永远都是字符串。如果你使用 string（字面量）以外的其他值作为属性名，那它首先会被转换为一个字符串。即使是数字也不例外，虽然在数组下标中使用的的确是数字，但是在对象属性名中数字会被转换成字符串，所以当心不要搞混对象和数组中数字的用法：

```
var myObject = { };

myObject[true] = "foo";
myObject[3] = "bar";
myObject[myObject] = "baz";

myObject["true"]; // "foo"
myObject["3"]; // "bar"
myObject["[object Object]"]; // "baz"
```

3.3.1 可计算属性名

如果你需要通过表达式来计算属性名，那么我们刚刚讲到的 myObject[..] 这种属性访问语法就可以派上用场了，如可以使用 myObject[prefix + name]。但是使用文字形式来声明对象时这样做是不行的。

ES6 增加了*可计算属性名*，可以在文字形式中使用 [] 包裹一个表达式来当作属性名：

```
var prefix = "foo";

var myObject = {
    [prefix + "bar"]:"hello",
    [prefix + "baz"]: "world"
};

myObject["foobar"]; // hello
myObject["foobaz"]; // world
```

可计算属性名最常用的场景可能是 ES6 的符号（Symbol），本书中不作详细介绍。不过简单来说，它们是一种新的基础数据类型，包含一个不透明且无法预测的值（从技术

角度来说就是一个字符串）。一般来说你不会用到符号的实际值（因为理论上来说在不同的 JavaScript 引擎中值是不同的），所以通常你接触到的是符号的名称，比如 Symbol.Something（这个名字是我编的）：

```
var myObject = {
    [Symbol.Something]: "hello world"
}
```

3.3.2 属性与方法

如果访问的对象属性是一个函数，有些开发者喜欢使用不一样的叫法以作区分。由于函数很容易被认为是属于某个对象，在其他语言中，属于对象（也被称为“类”）的函数通常被称为“方法”，因此把“属性访问”说成是“方法访问”也就不奇怪了。

有意思的是，JavaScript 的语法规范也做出了同样的区分。

从技术角度来说，函数永远不会“属于”一个对象，所以把对象内部引用的函数称为“方法”似乎有点不妥。

确实，有些函数具有 this 引用，有时候这些 this 确实会指向调用位置的对象引用。但是这种用法从本质上来说并没有把一个函数变成一个“方法”，因为 this 是在运行时根据调用位置动态绑定的，所以函数和对象的关系最多也只能说是间接关系。

无论返回值是什么类型，每次访问对象的属性就是属性访问。如果属性访问返回的是一个函数，那它也并不是一个“方法”。属性访问返回的函数和其他函数没有任何区别（除了可能发生的隐式绑定 this，就像我们刚才提到的）。

举例来说：

```
function foo() {
    console.log( "foo" );
}

var someFoo = foo; // 对 foo 的变量引用

var myObject = {
    someFoo: foo
};

foo; // function foo(){..}

someFoo; // function foo(){..}

myObject.someFoo; // function foo(){..}
```

someFoo 和 myObject.someFoo 只是对于同一个函数的不同引用，并不能说明这个函数是特别的或者“属于”某个对象。如果 foo() 定义时在内部有一个 this 引用，那这两个函数引

用的唯一区别就是 myObject.someFoo 中的 this 会被隐式绑定到一个对象。无论哪种引用形式都不能称之为“方法”。

或许有人会辩解说，函数并不是在定义时成为方法，而是在被调用时根据调用位置的不同（是否具有上下文对象——详见第 2 章）成为方法。即便如此，这种说法仍然有些不妥。

最保险的说法可能是，“函数”和“方法”在 JavaScript 中是可以互换的。

ES6 增加了 super 引用，一般来说会被用在 class 中（参见附录 A）。super 的行为似乎更有理由把 super 绑定的函数称为“方法”。但是再说一次，这些只是一些语义（和技术）上的微妙差别，本质是一样的。

即使你在对象的文字形式中声明一个函数表达式，这个函数也不会“属于”这个对象——它们只是对于相同函数对象的多个引用。

```
var myObject = {
    foo: function() {
        console.log( "foo" );
    }
};

var someFoo = myObject.foo;

someFoo; // function foo(){..}

myObject.foo; // function foo(){..}
```

第 6 章会介绍本例对象的文字形式中声明函数的语法，这是 ES6 增加的一种简易函数声明语法。

3.3.3 数组

数组也支持 [] 访问形式，不过就像我们之前提到过的，数组有一套更加结构化的值存储机制（不过仍然不限制值的类型）。数组期望的是数值下标，也就是说值存储的位置（通常被称为索引）是整数，比如说 0 和 42：

```
var myArray = [ "foo", 42, "bar" ];

myArray.length; // 3

myArray[0]; // "foo"

myArray[2]; // "bar"
```

数组也是对象，所以虽然每个下标都是整数，你仍然可以给数组添加属性：

```
var myArray = [ "foo", 42, "bar" ];

myArray.baz = "baz";

myArray.length; // 3

myArray.baz; // "baz"
```

可以看到虽然添加了命名属性（无论是通过 . 语法还是 [] 语法），数组的 length 值并未发生变化。

你完全可以把数组当作一个普通的键 / 值对象来使用，并且不添加任何数值索引，但是这并不是一个好主意。数组和普通的对象都根据其对应的行为和用途进行了优化，所以最好只用对象来存储键 / 值对，只用数组来存储数值下标 / 值对。

注意：如果你试图向数组添加一个属性，但是属性名“看起来”像一个数字，那它会变成一个数值下标（因此会修改数组的内容而不是添加一个属性）：

```
var myArray = [ "foo", 42, "bar" ];

myArray["3"] = "baz";

myArray.length; // 4

myArray[3]; // "baz"
```

3.3.4 复制对象

JavaScript 初学者最常见的问题之一就是如何复制一个对象。看起来应该有一个内置的 copy() 方法，是吧？实际上事情比你想象的更复杂，因为我们无法选择一个默认的复制算法。

举例来说，思考一下这个对象：

```
function anotherFunction() { /*..*/ }

var anotherObject = {
    c: true
};

var anotherArray = [];

var myObject = {
    a: 2,
    b: anotherObject, // 引用，不是复本！
    c: anotherArray, // 另一个引用！
    d: anotherFunction
};
```

```
anotherArray.push( anotherObject, myObject );
```

如何准确地表示 myObject 的复制呢？

首先，我们应该判断它是浅复制还是深复制。对于浅拷贝来说，复制出的新对象中 a 的值会复制旧对象中 a 的值，也就是 2，但是新对象中 b、c、d 三个属性其实只是三个引用，它们和旧对象中 b、c、d 引用的对象是一样的。对于深复制来说，除了复制 myObject 以外还会复制 anotherObject 和 anotherArray。这时问题就来了，anotherArray 引用了 anotherObject 和 myObject，所以又需要复制 myObject，这样就会由于循环引用导致死循环。

我们是应该检测循环引用并终止循环（不复制深层元素）？还是应当直接报错或者是选择其他方法？

除此之外，我们还不确定“复制”一个函数意味着什么。有些人会通过 toString() 来序列化一个函数的源代码（但是结果取决于 JavaScript 的具体实现，而且不同的引擎对于不同类型的函数处理方式并不完全相同）。

那么如何解决这些棘手问题呢？许多 JavaScript 框架都提出了自己的解决办法，但是 JavaScript 应当采用哪种方法作为标准呢？在很长一段时间里，这个问题都没有明确的答案。

对于 JSON 安全（也就是说可以被序列化为一个 JSON 字符串并且可以根据这个字符串解析出一个结构和值完全一样的对象）的对象来说，有一种巧妙的复制方法：

```
var newObj = JSON.parse( JSON.stringify( someObj ) );
```

当然，这种方法需要保证对象是 JSON 安全的，所以只适用于部分情况。

相比深复制，浅复制非常易懂并且问题要少得多，所以 ES6 定义了 Object.assign(..) 方法来实现浅复制。Object.assign(..) 方法的第一个参数是目标对象，之后还可以跟一个或多个源对象。它会遍历一个或多个源对象的所有可枚举（enumerable，参见下面的代码）的自有键（owned key，很快会介绍）并把它们复制（使用 = 操作符赋值）到目标对象，最后返回目标对象，就像这样：

```
var newObj = Object.assign( {}, myObject );

newObj.a; // 2
newObj.b === anotherObject; // true
newObj.c === anotherArray; // true
newObj.d === anotherFunction; // true
```

下一节会介绍“属性描述符”以及 Object.defineProperty(..) 的用法。但是需要注意的一点是，由于 Object.assign(..) 就是使用 = 操作符来赋值，所以源对象属性的一些特性（比如 writable）不会被复制到目标对象。

3.3.5 属性描述符

在 ES5 之前，JavaScript 语言本身并没有提供可以直接检测属性特性的方法，比如判断属性是否是只读。

但是从 ES5 开始，所有的属性都具备了属性描述符。

思考下面的代码：

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

Object.getOwnPropertyDescriptor( myObject, "a" );
// {
//    value: 2,
//    writable: true,
//    enumerable: true,
//    configurable: true
// }
```

如你所见，这个普通的对象属性对应的属性描述符（也被称为“数据描述符”，因为它只保存一个数据值）可不仅仅只是一个 2。它还包含另外三个特性：`writable`（可写）、`enumerable`（可枚举）和 `configurable`（可配置）。

在创建普通属性时属性描述符会使用默认值，我们也可以使用 `Object.defineProperty(..)` 来添加一个新属性或者修改一个已有属性（如果它是 `configurable`）并对特性进行设置。

举例来说：

```
var myObject = {};

Object.defineProperty( myObject, "a", {
    value: 2,
    writable: true,
    configurable: true,
    enumerable: true
} );

myObject.a; // 2
```

我们使用 `defineProperty(..)` 给 `myObject` 添加了一个普通的属性并显式指定了一些特性。然而，一般来说你不会使用这种方式，除非你想修改属性描述符。

1. Writable

`writable` 决定是否可以修改属性的值。

思考下面的代码：

```
var myObject = {};

Object.defineProperty( myObject, "a", {
    value: 2,
    writable: false, // 不可写!
    configurable: true,
    enumerable: true
} );

myObject.a = 3;

myObject.a; // 2
```

如你所见，我们对于属性值的修改静默失败（silently failed）了。如果在严格模式下，这种方法会出错：

```
"use strict";

var myObject = {};

Object.defineProperty( myObject, "a", {
    value: 2,
    writable: false, // 不可写!
    configurable: true,
    enumerable: true
} );

myObject.a = 3; // TypeError
```

`TypeError` 错误表示我们无法修改一个不可写的属性。

之后我们会介绍 getter 和 setter，不过简单来说，你可以把 `writable:false` 看作是属性不可改变，相当于你定义了一个空操作 setter。严格来说，如果要和 `writable:false` 一致的话，你的 setter 被调用时应当抛出一个 `TypeError` 错误。

2. Configurable

只要属性是可配置的，就可以使用 `defineProperty(..)` 方法来修改属性描述符：

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

myObject.a = 3;
myObject.a; // 3

Object.defineProperty( myObject, "a", {
    value: 4,
    writable: true,
    configurable: false, // 不可配置!
```

```
    enumerable: true
} );

myObject.a; // 4
myObject.a = 5;
myObject.a; // 5

Object.defineProperty( myObject, "a", {
    value: 6,
    writable: true,
    configurable: true,
    enumerable: true
} ); // TypeError
```

最后一个 defineProperty(..) 会产生一个 TypeError 错误，不管是不是处于严格模式，尝试修改一个不可配置的属性描述符都会出错。注意：如你所见，把 configurable 修改成 false 是单向操作，无法撤销！

要注意有一个小小的例外：即便属性是 configurable:false，我们还是可以把 writable 的状态由 true 改为 false，但是无法由 false 改为 true。

除了无法修改，configurable:false 还会禁止删除这个属性：

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

myObject.a; // 2

delete myObject.a;
myObject.a; // undefined

Object.defineProperty( myObject, "a", {
    value: 2,
    writable: true,
    configurable: false,
    enumerable: true
} );

myObject.a; // 2
delete myObject.a;
myObject.a; // 2
```

如你所见，最后一个 delete 语句（静默）失败了，因为属性是不可配置的。

在本例中，delete 只用来直接删除对象的（可删除）属性。如果对象的某个属性是某个对象 / 函数的最后一个*引用者*，对这个属性执行 delete 操作之后，这个未引用的对象 / 函

数就可以被垃圾回收。但是，不要把 delete 看作一个释放内存的工具（就像 C/C++ 中那样），它就是一个删除对象属性的操作，仅此而已。

3. Enumerable

这里我们要介绍的最后一个属性描述符（还有两个，我们会在介绍 getter 和 setter 时提到）是 enumerable。

从名字就可以看出，这个描述符控制的是属性是否会出现在对象的属性枚举中，比如说 for..in 循环。如果把 enumerable 设置成 false，这个属性就不会出现在枚举中，虽然仍然可以正常访问它。相对地，设置成 true 就会让它出现在枚举中。

用户定义的所有的普通属性默认都是 enumerable，这通常就是你想要的。但是如果你不希望某些特殊属性出现在枚举中，那就把它设置成 enumerable:false。

稍后我们会详细介绍可枚举性，这里先提示一下。

3.3.6 不变性

有时候你会希望属性或者对象是不可改变（无论有意还是无意）的，在 ES5 中可以通过很多种方法来实现。

很重要的一点是，所有的方法创建的都是浅不变形，也就是说，它们只会影响目标对象和它的直接属性。如果目标对象引用了其他对象（数组、对象、函数，等），其他对象的内容不受影响，仍然是可变的：

```
myImmutableObject.foo; // [1,2,3]
myImmutableObject.foo.push( 4 );
myImmutableObject.foo; // [1,2,3,4]
```

假设代码中的 myImmutableObject 已经被创建而且是不可变的，但是为了保护它的内容 myImmutableObject.foo，你还需要使用下面的方法让 foo 也不可变。

在 JavaScript 程序中很少需要深不可变性。有些特殊情况可能需要这样做，但是根据通用的设计模式，如果你发现需要密封或者冻结所有的对象，那你或许应当退一步，重新思考一下程序的设计，让它能更好地应对对象值的改变。

1. 对象常量

结合 writable:false 和 configurable:false 就可以创建一个真正的常量属性（不可修改、重定义或者删除）：

```
var myObject = {};

Object.defineProperty( myObject, "FAVORITE_NUMBER", {
    value: 42,
    writable: false,
    configurable: false
} );
```

2. 禁止扩展

如果你想禁止一个对象添加新属性并且保留已有属性，可以使用 `Object.preventExtensions(..)`：

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

Object.preventExtensions( myObject );

myObject.b = 3;
myObject.b; // undefined
```

在非严格模式下，创建属性 `b` 会静默失败。在严格模式下，将会抛出 `TypeError` 错误。

3. 密封

`Object.seal(..)` 会创建一个“密封”的对象，这个方法实际上会在一个现有对象上调用 `Object.preventExtensions(..)` 并把所有现有属性标记为 `configurable:false`。

所以，密封之后不仅不能添加新属性，也不能重新配置或者删除任何现有属性（虽然可以修改属性的值）。

4. 冻结

`Object.freeze(..)` 会创建一个冻结对象，这个方法实际上会在一个现有对象上调用 `Object.seal(..)` 并把所有“数据访问”属性标记为 `writable:false`，这样就无法修改它们的值。

这个方法是你可以应用在对象上的级别最高的不可变性，它会禁止对于对象本身及其任意直接属性的修改（不过就像我们之前说过的，这个对象引用的其他对象是不受影响的）。

你可以“深度冻结”一个对象，具体方法为，首先在这个对象上调用 `Object.freeze(..)`，然后遍历它引用的所有对象并在这些对象上调用 `Object.freeze(..)`。但是一定要小心，因为这样做有可能会在无意中冻结其他（共享）对象。

3.3.7 [[Get]]

属性访问在实现时有一个微妙却非常重要的细节，思考下面的代码：