8.1.1　属性的类型

ECMA-262 使用一些内部特性来描述属性的特征。这些特性是由为 JavaScript 实现引擎的规范定义的。因此，开发者不能在 JavaScript 中直接访问这些特性。为了将某个特性标识为内部特性，规范会用两个中括号把特性的名称括起来，比如[[Enumerable]]。

属性分两种：数据属性和访问器属性。

1. 数据属性

数据属性包含一个保存数据值的位置。值会从这个位置读取，也会写入到这个位置。数据属性有 4 个特性描述它们的行为。

- [[Configurable]]：表示属性是否可以通过 delete 删除并重新定义，是否可以修改它的特性，以及是否可以把它改为访问器属性。默认情况下，所有直接定义在对象上的属性的这个特性都是 true，如前面的例子所示。
- [[Enumerable]]：表示属性是否可以通过 for-in 循环返回。默认情况下，所有直接定义在对象上的属性的这个特性都是 true，如前面的例子所示。
- [[Writable]]：表示属性的值是否可以被修改。默认情况下，所有直接定义在对象上的属性的这个特性都是 true，如前面的例子所示。
- [[Value]]：包含属性实际的值。这就是前面提到的那个读取和写入属性值的位置。这个特性的默认值为 undefined。

在像前面例子中那样将属性显式添加到对象之后，[[Configurable]]、[[Enumerable]]和[[Writable]]都会被设置为 true，而[[Value]]特性会被设置为指定的值。比如：

```
let person = {
  name: "Nicholas"
};
```

这里，我们创建了一个名为 name 的属性，并给它赋予了一个值"Nicholas"。这意味着[[Value]]特性会被设置为"Nicholas"，之后对这个值的任何修改都会保存这个位置。

要修改属性的默认特性，就必须使用 Object.defineProperty()方法。这个方法接收 3 个参数：要给其添加属性的对象、属性的名称和一个描述符对象。最后一个参数，即描述符对象上的属性可以包含：configurable、enumerable、writable 和 value，跟相关特性的名称一一对应。根据要修改的特性，可以设置其中一个或多个值。比如：

```
let person = {};
Object.defineProperty(person, "name", {
  writable: false,
  value: "Nicholas"
});
console.log(person.name); // "Nicholas"
person.name = "Greg";
console.log(person.name); // "Nicholas"
```

这个例子创建了一个名为 name 的属性并给它赋予了一个只读的值"Nicholas"。这个属性的值就不能再修改了，在非严格模式下尝试给这个属性重新赋值会被忽略。在严格模式下，尝试修改只读属性的值会抛出错误。

类似的规则也适用于创建不可配置的属性。比如：

```
let person = {};
Object.defineProperty(person, "name", {
  configurable: false,
  value: "Nicholas"
});
console.log(person.name); // "Nicholas"
delete person.name;
console.log(person.name); // "Nicholas"
```

这个例子把 configurable 设置为 false，意味着这个属性不能从对象上删除。非严格模式下对这个属性调用 delete 没有效果，严格模式下会抛出错误。此外，一个属性被定义为不可配置之后，就不能再变回可配置的了。再次调用 Object.defineProperty()并修改任何非 writable 属性会导致错误：

```
let person = {};
Object.defineProperty(person, "name", {
  configurable: false,
  value: "Nicholas"
});

// 抛出错误
Object.defineProperty(person, "name", {
  configurable: true,
  value: "Nicholas"
});
```

因此，虽然可以对同一个属性多次调用 Object.defineProperty()，但在把 configurable 设置为 false 之后就会受限制了。

在调用 Object.defineProperty()时，configurable、enumerable 和 writable 的值如果不指定，则都默认为 false。多数情况下，可能都不需要 Object.defineProperty()提供的这些强大的设置，但要理解 JavaScript 对象，就要理解这些概念。

2. 访问器属性

访问器属性不包含数据值。相反，它们包含一个获取（getter）函数和一个设置（setter）函数，不过这两个函数不是必需的。在读取访问器属性时，会调用获取函数，这个函数的责任就是返回一个有效的值。在写入访问器属性时，会调用设置函数并传入新值，这个函数必须决定对数据做出什么修改。访问器属性有 4 个特性描述它们的行为。

- [[Configurable]]：表示属性是否可以通过 delete 删除并重新定义，是否可以修改它的特性，以及是否可以把它改为数据属性。默认情况下，所有直接定义在对象上的属性的这个特性都是 true。
- [[Enumerable]]：表示属性是否可以通过 for-in 循环返回。默认情况下，所有直接定义在对象上的属性的这个特性都是 true。
- [[Get]]：获取函数，在读取属性时调用。默认值为 undefined。
- [[Set]]：设置函数，在写入属性时调用。默认值为 undefined。

访问器属性是不能直接定义的，必须使用 Object.defineProperty()。下面是一个例子：

```
// 定义一个对象，包含伪私有成员 year_和公共成员 edition
let book = {
  year_: 2017,
  edition: 1
```

```
};

Object.defineProperty(book, "year", {
  get() {
    return this.year_;
  },
  set(newValue) {
    if (newValue > 2017) {
      this.year_ = newValue;
      this.edition += newValue - 2017;
    }
  }
});
book.year = 2018;
console.log(book.edition); // 2
```

在这个例子中，对象 book 有两个默认属性：year_和 edition。year_中的下划线常用来表示该属性并不希望在对象方法的外部被访问。另一个属性 year 被定义为一个访问器属性，其中获取函数简单地返回 year_的值，而设置函数会做一些计算以决定正确的版本（edition）。因此，把 year 属性修改为 2018 会导致 year_变成 2018，edition 变成 2。这是访问器属性的典型使用场景，即设置一个属性值会导致一些其他变化发生。

获取函数和设置函数不一定都要定义。只定义获取函数意味着属性是只读的，尝试修改属性会被忽略。在严格模式下，尝试写入只定义了获取函数的属性会抛出错误。类似地，只有一个设置函数的属性是不能读取的，非严格模式下读取会返回 undefined，严格模式下会抛出错误。

在不支持 Object.defineProperty()的浏览器中没有办法修改[[Configurable]]或[[Enumerable]]。

> **注意**　在 ECMAScript 5 以前，开发者会使用两个非标准的访问创建访问器属性：__defineGetter__()和__defineSetter__()。这两个方法最早是 Firefox 引入的，后来 Safari、Chrome 和 Opera 也实现了。

8.1.2　定义多个属性

在一个对象上同时定义多个属性的可能性是非常大的。为此，ECMAScript 提供了 Object.defineProperties()方法。这个方法可以通过多个描述符一次性定义多个属性。它接收两个参数：要为之添加或修改属性的对象和另一个描述符对象，其属性与要添加或修改的属性一一对应。比如：

```
let book = {};
Object.defineProperties(book, {
  year_: {
    value: 2017
  },

  edition: {
    value: 1
  },

  year: {
    get() {
      return this.year_;
    },
```

```
    set(newValue) {
      if (newValue > 2017) {
        this.year_ = newValue;
        this.edition += newValue - 2017;
      }
    }
  }
});
```

这段代码在 book 对象上定义了两个数据属性 year_和 edition，还有一个访问器属性 year。最终的对象跟上一节示例中的一样。唯一的区别是所有属性都是同时定义的，并且数据属性的 configurable、enumerable 和 writable 特性值都是 false。

8.1.3 读取属性的特性

使用 Object.getOwnPropertyDescriptor()方法可以取得指定属性的属性描述符。这个方法接收两个参数：属性所在的对象和要取得其描述符的属性名。返回值是一个对象，对于访问器属性包含 configurable、enumerable、get 和 set 属性，对于数据属性包含 configurable、enumerable、writable 和 value 属性。比如：

```
let book = {};
Object.defineProperties(book, {
  year_: {
    value: 2017
  },

  edition: {
    value: 1
  },

  year: {
    get: function() {
      return this.year_;
    },

    set: function(newValue){
      if (newValue > 2017) {
        this.year_ = newValue;
        this.edition += newValue - 2017;
      }
    }
  }
});

let descriptor = Object.getOwnPropertyDescriptor(book, "year_");
console.log(descriptor.value);          // 2017
console.log(descriptor.configurable);   // false
console.log(typeof descriptor.get);     // "undefined"
let descriptor = Object.getOwnPropertyDescriptor(book, "year");
console.log(descriptor.value);          // undefined
console.log(descriptor.enumerable);     // false
console.log(typeof descriptor.get);     // "function"
```

对于数据属性 year_，value 等于原来的值，configurable 是 false，get 是 undefined。对于访问器属性 year，value 是 undefined，enumerable 是 false，get 是一个指向获取函数的指针。

ECMAScript 2017 新增了 `Object.getOwnPropertyDescriptors()`静态方法。这个方法实际上会在每个自有属性上调用 `Object.getOwnPropertyDescriptor()`并在一个新对象中返回它们。对于前面的例子，使用这个静态方法会返回如下对象：

```
let book = {};
Object.defineProperties(book, {
  year_: {
    value: 2017
  },

  edition: {
    value: 1
  },

  year: {
    get: function() {
      return this.year_;
    },

    set: function(newValue){
      if (newValue > 2017) {
        this.year_ = newValue;
        this.edition += newValue - 2017;
      }
    }
  }
});

console.log(Object.getOwnPropertyDescriptors(book));
// {
//   edition: {
//     configurable: false,
//     enumerable: false,
//     value: 1,
//     writable: false
//   },
//   year: {
//     configurable: false,
//     enumerable: false,
//     get: f(),
//     set: f(newValue),
//   },
//   year_: {
//     configurable: false,
//     enumerable: false,
//     value: 2017,
//     writable: false
//   }
// }
```

8.1.4　合并对象

JavaScript 开发者经常觉得“合并”（merge）两个对象很有用。更具体地说，就是把源对象所有的本地属性一起复制到目标对象上。有时候这种操作也被称为“混入”（mixin），因为目标对象通过混入源对象的属性得到了增强。

ECMAScript 6 专门为合并对象提供了 Object.assign()方法。这个方法接收一个目标对象和一个或多个源对象作为参数，然后将每个源对象中可枚举（Object.propertyIsEnumerable()返回 true）和自有（Object.hasOwnProperty()返回 true）属性复制到目标对象。以字符串和符号为键的属性会被复制。对每个符合条件的属性，这个方法会使用源对象上的[[Get]]取得属性的值，然后使用目标对象上的[[Set]]设置属性的值。

```
let dest, src, result;

/**
 * 简单复制
 */
dest = {};
src = { id: 'src' };

result = Object.assign(dest, src);

// Object.assign 修改目标对象
// 也会返回修改后的目标对象
console.log(dest === result); // true
console.log(dest !== src);    // true
console.log(result);          // { id: src }
console.log(dest);            // { id: src }


/**
 * 多个源对象
 */
dest = {};

result = Object.assign(dest, { a: 'foo' }, { b: 'bar' });

console.log(result); // { a: foo, b: bar }


/**
 * 获取函数与设置函数
 */
dest = {
  set a(val) {
    console.log(`Invoked dest setter with param ${val}`);
  }
};
src = {
  get a() {
    console.log('Invoked src getter');
    return 'foo';
  }
};

Object.assign(dest, src);
// 调用 src 的获取方法
// 调用 dest 的设置方法并传入参数"foo"
// 因为这里的设置函数不执行赋值操作
// 所以实际上并没有把值转移过来
console.log(dest); // { set a(val) {...} }
```

Object.assign()实际上对每个源对象执行的是浅复制。如果多个源对象都有相同的属性，则使用最后一个复制的值。此外，从源对象访问器属性取得的值，比如获取函数，会作为一个静态值赋给目标对象。换句话说，不能在两个对象间转移获取函数和设置函数。

```
let dest, src, result;

/**
 * 覆盖属性
 */
dest = { id: 'dest' };

result = Object.assign(dest, { id: 'src1', a: 'foo' }, { id: 'src2', b: 'bar' });

// Object.assign 会覆盖重复的属性
console.log(result); // { id: src2, a: foo, b: bar }

// 可以通过目标对象上的设置函数观察到覆盖的过程：
dest = {
  set id(x) {
    console.log(x);
  }
};

Object.assign(dest, { id: 'first' }, { id: 'second' }, { id: 'third' });
// first
// second
// third


/**
 * 对象引用
 */

dest = {};
src = { a: {} };

Object.assign(dest, src);

// 浅复制意味着只会复制对象的引用
console.log(dest);               // { a :{} }
console.log(dest.a === src.a);  // true
```

如果赋值期间出错，则操作会中止并退出，同时抛出错误。Object.assign()没有“回滚”之前赋值的概念，因此它是一个尽力而为、可能只会完成部分复制的方法。

```
let dest, src, result;

/**
 * 错误处理
 */
dest = {};
src = {
  a: 'foo',
  get b() {
    // Object.assign()在调用这个获取函数时会抛出错误
    throw new Error();
  },
```

```
  c: 'bar'
};

try {
  Object.assign(dest, src);
} catch(e) {}

// Object.assign()没办法回滚已经完成的修改
// 因此在抛出错误之前，目标对象上已经完成的修改会继续存在：
console.log(dest); // { a: foo }
```

8.1.5 对象标识及相等判定

在 ECMAScript 6 之前，有些特殊情况即使是===操作符也无能为力：

```
// 这些是===符合预期的情况
console.log(true === 1);  // false
console.log({} === {});   // false
console.log("2" === 2);   // false

// 这些情况在不同 JavaScript 引擎中表现不同，但仍被认为相等
console.log(+0 === -0);   // true
console.log(+0 === 0);    // true
console.log(-0 === 0);    // true

// 要确定 NaN 的相等性，必须使用极为讨厌的 isNaN()
console.log(NaN === NaN); // false
console.log(isNaN(NaN));  // true
```

为改善这类情况，ECMAScript 6 规范新增了 `Object.is()`，这个方法与===很像，但同时也考虑到了上述边界情形。这个方法必须接收两个参数：

```
console.log(Object.is(true, 1));  // false
console.log(Object.is({}, {}));   // false
console.log(Object.is("2", 2));   // false

// 正确的 0、-0、+0 相等/不等判定
console.log(Object.is(+0, -0));   // false
console.log(Object.is(+0, 0));    // true
console.log(Object.is(-0, 0));    // false

// 正确的 NaN 相等判定
console.log(Object.is(NaN, NaN)); // true
```

要检查超过两个值，递归地利用相等性传递即可：

```
function recursivelyCheckEqual(x, ...rest) {
  return Object.is(x, rest[0]) &&
         (rest.length < 2 || recursivelyCheckEqual(...rest));
}
```

8.1.6 增强的对象语法

ECMAScript 6 为定义和操作对象新增了很多极其有用的语法糖特性。这些特性都没有改变现有引擎的行为，但极大地提升了处理对象的方便程度。

本节介绍的所有对象语法同样适用于 ECMAScript 6 的类，本章后面会讨论。

> **注意**　相比于以往的替代方案，本节介绍的增强对象语法可以说是一骑绝尘。因此本章及本书会默认使用这些新语法特性。

1. 属性值简写

在给对象添加变量的时候，开发者经常会发现属性名和变量名是一样的。例如：

```
let name = 'Matt';

let person = {
  name: name
};

console.log(person); // { name: 'Matt' }
```

为此，简写属性名语法出现了。简写属性名只要使用变量名（不用再写冒号）就会自动被解释为同名的属性键。如果没有找到同名变量，则会抛出 ReferenceError。

以下代码和之前的代码是等价的：

```
let name = 'Matt';

let person = {
  name
};

console.log(person); // { name: 'Matt' }
```

代码压缩程序会在不同作用域间保留属性名，以防止找不到引用。以下面的代码为例：

```
function makePerson(name) {
  return {
    name
  };
}

let person = makePerson('Matt');

console.log(person.name);  // Matt
```

在这里，即使参数标识符只限定于函数作用域，编译器也会保留初始的 name 标识符。如果使用 Google Closure 编译器压缩，那么函数参数会被缩短，而属性名不变：

```
function makePerson(a) {
  return {
    name: a
  };
}

var person = makePerson("Matt");

console.log(person.name); // Matt
```

2. 可计算属性

在引入可计算属性之前，如果想使用变量的值作为属性，那么必须先声明对象，然后使用中括号语法来添加属性。换句话说，不能在对象字面量中直接动态命名属性。比如：

```
const nameKey = 'name';
const ageKey = 'age';
```

```
const jobKey = 'job';

let person = {};
person[nameKey] = 'Matt';
person[ageKey] = 27;
person[jobKey] = 'Software engineer';

console.log(person); // { name: 'Matt', age: 27, job: 'Software engineer' }
```

有了可计算属性，就可以在对象字面量中完成动态属性赋值。中括号包围的对象属性键告诉运行时将其作为 JavaScript 表达式而不是字符串来求值：

```
const nameKey = 'name';
const ageKey = 'age';
const jobKey = 'job';

let person = {
  [nameKey]: 'Matt',
  [ageKey]: 27,
  [jobKey]: 'Software engineer'
};

console.log(person); // { name: 'Matt', age: 27, job: 'Software engineer' }
```

因为被当作 JavaScript 表达式求值，所以可计算属性本身可以是复杂的表达式，在实例化时再求值：

```
const nameKey = 'name';
const ageKey = 'age';
const jobKey = 'job';
let uniqueToken = 0;

function getUniqueKey(key) {
  return `${key}_${uniqueToken++}`;
}

let person = {
  [getUniqueKey(nameKey)]: 'Matt',
  [getUniqueKey(ageKey)]: 27,
  [getUniqueKey(jobKey)]: 'Software engineer'
};

console.log(person);  // { name_0: 'Matt', age_1: 27, job_2: 'Software engineer' }
```

> **注意** 可计算属性表达式中抛出任何错误都会中断对象创建。如果计算属性的表达式有副作用，那就要小心了，因为如果表达式抛出错误，那么之前完成的计算是不能回滚的。

3. 简写方法名

在给对象定义方法时，通常都要写一个方法名、冒号，然后再引用一个匿名函数表达式，如下所示：

```
let person = {
  sayName: function(name) {
    console.log(`My name is ${name}`);
  }
};

person.sayName('Matt'); // My name is Matt
```