第6章 集合引用类型

本章内容

- 对象
- 数组与定型数组
- Map、WeakMap、Set 以及 WeakSet 类型

6.1 Object

到目前为止，大多数引用值的示例使用的是 Object 类型。Object 是 ECMAScript 中最常用的类型之一。虽然 Object 的实例没有多少功能，但很适合存储和在应用程序间交换数据。

显式地创建 Object 的实例有两种方式。第一种是使用 new 操作符和 Object 构造函数，如下所示：

```
let person = new Object();
person.name = "Nicholas";
person.age = 29;
```

另一种方式是使用**对象字面量**（object literal）表示法。对象字面量是对象定义的简写形式，目的是为了简化包含大量属性的对象的创建。比如，下面的代码定义了与前面示例相同的 person 对象，但使用的是对象字面量表示法：

```
let person = {
  name: "Nicholas",
  age: 29
};
```

在这个例子中，左大括号（{）表示对象字面量开始，因为它出现在一个**表达式上下文**（expression context）中。在 ECMAScript 中，表达式上下文指的是期待返回值的上下文。赋值操作符表示后面要期待一个值，因此左大括号表示一个表达式的开始。同样是左大括号，如果出现在**语句上下文**（statement context）中，比如 if 语句的条件后面，则表示一个语句块的开始。

接下来指定了 name 属性，后跟一个冒号，然后是属性的值。逗号用于在对象字面量中分隔属性，因此字符串"Nicholas"后面有一个逗号，而 29 后面没有，因为 age 是这个对象的最后一个属性。在最后一个属性后面加上逗号在非常老的浏览器中会导致报错，但所有现代浏览器都支持这种写法。

在对象字面量表示法中，属性名可以是字符串或数值，比如：

```
let person = {
  "name": "Nicholas",
  "age": 29,
  5: true
};
```

这个例子会得到一个带有属性 name、age 和 5 的对象。注意，数值属性会自动转换为字符串。

当然也可以用对象字面量表示法来定义一个只有默认属性和方法的对象，只要使用一对大括号，中间留空就行了：

```
let person = {}; // 与 new Object()相同
person.name = "Nicholas";
person.age = 29;
```

这个例子跟本节开始的第一个例子是等效的，虽然看起来有点怪。对象字面量表示法通常只在为了让属性一目了然时才使用。

> **注意** 在使用对象字面量表示法定义对象时，并不会实际调用 Object 构造函数。

虽然使用哪种方式创建 Object 实例都可以，但实际上开发者更倾向于使用对象字面量表示法。这是因为对象字面量代码更少，看起来也更有封装所有相关数据的感觉。事实上，对象字面量已经成为给函数传递大量可选参数的主要方式，比如：

```
function displayInfo(args) {
  let output = "";

  if (typeof args.name == "string"){
    output += "Name: " + args.name + "\n";
  }

  if (typeof args.age == "number") {
    output += "Age: " + args.age + "\n";
  }

  alert(output);
}

displayInfo({
  name: "Nicholas",
  age: 29
});

displayInfo({
  name: "Greg"
});
```

这里，函数 displayInfo()接收一个名为 args 的参数。这个参数可能有属性 name 或 age，也可能两个属性都有或者都没有。函数内部会使用 typeof 操作符测试每个属性是否存在，然后根据属性有无构造并显示一条消息。然后，这个函数被调用了两次，每次都通过一个对象字面量传入了不同的数据。两种情况下，函数都正常运行。

> **注意** 这种模式非常适合函数有大量可选参数的情况。一般来说，命名参数更直观，但在可选参数过多的时候就显得笨拙了。最好的方式是对必选参数使用命名参数，再通过一个对象字面量来封装多个可选参数。

虽然属性一般是通过**点语法**来存取的，这也是面向对象语言的惯例，但也可以使用中括号来存取属

性。在使用中括号时，要在括号内使用属性名的字符串形式，比如：

```
console.log(person["name"]); // "Nicholas"
console.log(person.name);    // "Nicholas"
```

从功能上讲，这两种存取属性的方式没有区别。使用中括号的主要优势就是可以通过变量访问属性，就像下面这个例子中一样：

```
let propertyName = "name";
console.log(person[propertyName]); // "Nicholas"
```

另外，如果属性名中包含可能会导致语法错误的字符，或者包含关键字/保留字时，也可以使用中括号语法。比如：

```
person["first name"] = "Nicholas";
```

因为"first name"中包含一个空格，所以不能使用点语法来访问。不过，属性名中是可以包含非字母数字字符的，这时候只要用中括号语法存取它们就行了。

通常，点语法是首选的属性存取方式，除非访问属性时必须使用变量。

> **注意**　第 8 章将更全面、深入地介绍 Object 类型。

6.2　Array

除了 Object，Array 应该就是 ECMAScript 中最常用的类型了。ECMAScript 数组跟其他编程语言的数组有很大区别。跟其他语言中的数组一样，ECMAScript 数组也是一组有序的数据，但跟其他语言不同的是，数组中每个槽位可以存储任意类型的数据。这意味着可以创建一个数组，它的第一个元素是字符串，第二个元素是数值，第三个是对象。ECMAScript 数组也是动态大小的，会随着数据添加而自动增长。

6.2.1　创建数组

有几种基本的方式可以创建数组。一种是使用 Array 构造函数，比如：

```
let colors = new Array();
```

如果知道数组中元素的数量，那么可以给构造函数传入一个数值，然后 length 属性就会被自动创建并设置为这个值。比如，下面的代码会创建一个初始 length 为 20 的数组：

```
let colors = new Array(20);
```

也可以给 Array 构造函数传入要保存的元素。比如，下面的代码会创建一个包含 3 个字符串值的数组：

```
let colors = new Array("red", "blue", "green");
```

创建数组时可以给构造函数传一个值。这时候就有点问题了，因为如果这个值是数值，则会创建一个长度为指定数值的数组；而如果这个值是其他类型的，则会创建一个只包含该特定值的数组。下面看一个例子：

```
let colors = new Array(3);      // 创建一个包含 3 个元素的数组
let names = new Array("Greg"); // 创建一个只包含一个元素，即字符串"Greg"的数组
```

在使用 Array 构造函数时，也可以省略 new 操作符。结果是一样的，比如：

```
let colors = Array(3);     // 创建一个包含 3 个元素的数组
let names = Array("Greg"); // 创建一个只包含一个元素，即字符串"Greg"的数组
```

另一种创建数组的方式是使用**数组字面量**（array literal）表示法。数组字面量是在中括号中包含以逗号分隔的元素列表，如下面的例子所示：

```
let colors = ["red", "blue", "green"];  // 创建一个包含 3 个元素的数组
let names = [];                         // 创建一个空数组
let values = [1,2,];                    // 创建一个包含 2 个元素的数组
```

在这个例子中，第一行创建一个包含 3 个字符串的数组。第二行用一对空中括号创建了一个空数组。第三行展示了在数组最后一个值后面加逗号的效果：values 是一个包含两个值（1 和 2）的数组。

> **注意** 与对象一样，在使用数组字面量表示法创建数组不会调用 Array 构造函数。

Array 构造函数还有两个 ES6 新增的用于创建数组的静态方法：from()和 of()。from()用于将类数组结构转换为数组实例，而 of()用于将一组参数转换为数组实例。

Array.from()的第一个参数是一个类数组对象，即任何可迭代的结构，或者有一个 length 属性和可索引元素的结构。这种方式可用于很多场合：

```
// 字符串会被拆分为单字符数组
console.log(Array.from("Matt")); // ["M", "a", "t", "t"]

// 可以使用 from()将集合和映射转换为一个新数组
const m = new Map().set(1, 2)
                   .set(3, 4);
const s = new Set().add(1)
                   .add(2)
                   .add(3)
                   .add(4);

console.log(Array.from(m)); // [[1, 2], [3, 4]]
console.log(Array.from(s)); // [1, 2, 3, 4]

// Array.from()对现有数组执行浅复制
const a1 = [1, 2, 3, 4];
const a2 = Array.from(a1);

console.log(a1);        // [1, 2, 3, 4]
alert(a1 === a2); // false


// 可以使用任何可迭代对象
const iter = {
  *[Symbol.iterator]() {
    yield 1;
    yield 2;
    yield 3;
    yield 4;

  }
};
console.log(Array.from(iter)); // [1, 2, 3, 4]
```

```
// arguments 对象可以被轻松地转换为数组
function getArgsArray() {
  return Array.from(arguments);
}
console.log(getArgsArray(1, 2, 3, 4)); // [1, 2, 3, 4]

// from()也能转换带有必要属性的自定义对象
const arrayLikeObject = {
  0: 1,
  1: 2,
  2: 3,
  3: 4,
  length: 4
};
console.log(Array.from(arrayLikeObject)); // [1, 2, 3, 4]
```

`Array.from()`还接收第二个可选的映射函数参数。这个函数可以直接增强新数组的值，而无须像调用 `Array.from().map()`那样先创建一个中间数组。还可以接收第三个可选参数，用于指定映射函数中 `this` 的值。但这个重写的 `this` 值在箭头函数中不适用。

```
const a1 = [1, 2, 3, 4];
const a2 = Array.from(a1, x => x**2);
const a3 = Array.from(a1, function(x) {return x**this.exponent}, {exponent: 2});
console.log(a2);  // [1, 4, 9, 16]
console.log(a3);  // [1, 4, 9, 16]
```

`Array.of()`可以把一组参数转换为数组。这个方法用于替代在 ES6 之前常用的 `Array.prototype.slice.call(arguments)`，一种异常笨拙的将 `arguments` 对象转换为数组的写法：

```
console.log(Array.of(1, 2, 3, 4)); // [1, 2, 3, 4]
console.log(Array.of(undefined));  // [undefined]
```

6.2.2　数组空位

使用数组字面量初始化数组时，可以使用一串逗号来创建空位（hole）。ECMAScript 会将逗号之间相应索引位置的值当成空位，ES6 规范重新定义了该如何处理这些空位。

可以像下面这样创建一个空位数组：

```
const options = [,,,,,]; // 创建包含 5 个元素的数组
console.log(options.length);   // 5
console.log(options);          // [,,,,,]
```

ES6 新增的方法和迭代器与早期 ECMAScript 版本中存在的方法行为不同。ES6 新增方法普遍将这些空位当成存在的元素，只不过值为 `undefined`：

```
const options = [1,,,,5];

for (const option of options) {
  console.log(option === undefined);
}
// false
// true
// true
// true
// false
```

```
const a = Array.from([,,,]); // 使用 ES6 的 Array.from()创建的包含 3 个空位的数组
for (const val of a) {
  alert(val === undefined);
}
// true
// true
// true

alert(Array.of(...[,,,])); // [undefined, undefined, undefined]

for (const [index, value] of options.entries()) {
  alert(value);
}
// 1
// undefined
// undefined
// undefined
// 5
```

ES6 之前的方法则会忽略这个空位，但具体的行为也会因方法而异：

```
const options = [1,,,,5];

// map()会跳过空位置
console.log(options.map(() => 6));  // [6, undefined, undefined, undefined, 6]

// join()视空位置为空字符串
console.log(options.join('-'));     // "1----5"
```

> **注意** 由于行为不一致和存在性能隐患，因此实践中要避免使用数组空位。如果确实需要空位，则可以显式地用 undefined 值代替。

6.2.3 数组索引

要取得或设置数组的值，需要使用中括号并提供相应值的数字索引，如下所示：

```
let colors = ["red", "blue", "green"];  // 定义一个字符串数组
alert(colors[0]);                       // 显示第一项
colors[2] = "black";                    // 修改第三项
colors[3] = "brown";                    // 添加第四项
```

在中括号中提供的索引表示要访问的值。如果索引小于数组包含的元素数，则返回存储在相应位置的元素，就像示例中 colors[0]显示"red"一样。设置数组的值方法也是一样的，就是替换指定位置的值。如果把一个值设置给超过数组最大索引的索引，就像示例中的 colors[3]，则数组长度会自动扩展到该索引值加 1（示例中设置的索引 3，所以数组长度变成了 4）。

数组中元素的数量保存在 length 属性中，这个属性始终返回 0 或大于 0 的值，如下例所示：

```
let colors = ["red", "blue", "green"];  // 创建一个包含 3 个字符串的数组
let names = [];                         // 创建一个空数组

alert(colors.length); // 3
alert(names.length);  // 0
```

数组 length 属性的独特之处在于，它不是只读的。通过修改 length 属性，可以从数组末尾删除

或添加元素。来看下面的例子：

```
let colors = ["red", "blue", "green"]; // 创建一个包含 3 个字符串的数组
colors.length = 2;
alert(colors[2]);  // undefined
```

这里，数组 colors 一开始有 3 个值。将 length 设置为 2，就删除了最后一个（位置 2 的）值，因此 colors[2]就没有值了。如果将 length 设置为大于数组元素数的值，则新添加的元素都将以 undefined 填充，如下例所示：

```
let colors = ["red", "blue", "green"];  // 创建一个包含 3 个字符串的数组
colors.length = 4;
alert(colors[3]);  // undefined
```

这里将数组 colors 的 length 设置为 4，虽然数组只包含 3 个元素。位置 3 在数组中不存在，因此访问其值会返回特殊值 undefined。

使用 length 属性可以方便地向数组末尾添加元素，如下例所示：

```
let colors = ["red", "blue", "green"];  // 创建一个包含 3 个字符串的数组
colors[colors.length] = "black";        // 添加一种颜色（位置 3）
colors[colors.length] = "brown";        // 再添加一种颜色（位置 4）
```

数组中最后一个元素的索引始终是 length - 1，因此下一个新增槽位的索引就是 length。每次在数组最后一个元素后面新增一项，数组的 length 属性都会自动更新，以反映变化。这意味着第二行的 colors[colors.length]会在位置 3 添加一个新元素，下一行则会在位置 4 添加一个新元素。新的长度会在新增元素被添加到当前数组外部的位置上时自动更新。换句话说，就是 length 属性会更新为位置加上 1，如下例所示：

```
let colors = ["red", "blue", "green"];  // 创建一个包含 3 个字符串的数组
colors[99] = "black";                   // 添加一种颜色（位置 99）
alert(colors.length);                   // 100
```

这里，colors 数组有一个值被插入到位置 99，结果新 length 就变成了 100（99 + 1）。这中间的所有元素，即位置 3~98，实际上并不存在，因此在访问时会返回 undefined。

> **注意**　数组最多可以包含 4 294 967 295 个元素，这对于大多数编程任务应该足够了。如果尝试添加更多项，则会导致抛出错误。以这个最大值作为初始值创建数组，可能导致脚本运行时间过长的错误。

6.2.4　检测数组

一个经典的 ECMAScript 问题是判断一个对象是不是数组。在只有一个网页（因而只有一个全局作用域）的情况下，使用 instanceof 操作符就足矣：

```
if (value instanceof Array){
  // 操作数组
}
```

使用 instanceof 的问题是假定只有一个全局执行上下文。如果网页里有多个框架，则可能涉及两个不同的全局执行上下文，因此就会有两个不同版本的 Array 构造函数。如果要把数组从一个框架传给另一个框架，则这个数组的构造函数将有别于在第二个框架内本地创建的数组。

为解决这个问题，ECMAScript 提供了 Array.isArray()方法。这个方法的目的就是确定一个值是否为数组，而不用管它是在哪个全局执行上下文中创建的。来看下面的例子：

```
if (Array.isArray(value)){
  // 操作数组
}
```

6.2.5 迭代器方法

在 ES6 中，Array 的原型上暴露了 3 个用于检索数组内容的方法：keys()、values()和 entries()。keys()返回数组索引的迭代器，values()返回数组元素的迭代器，而 entries()返回索引/值对的迭代器：

```
const a = ["foo", "bar", "baz", "qux"];

// 因为这些方法都返回迭代器，所以可以将它们的内容
// 通过 Array.from()直接转换为数组实例
const aKeys = Array.from(a.keys());
const aValues = Array.from(a.values());
const aEntries = Array.from(a.entries());

console.log(aKeys);     // [0, 1, 2, 3]
console.log(aValues);   // ["foo", "bar", "baz", "qux"]
console.log(aEntries);  // [[0, "foo"], [1, "bar"], [2, "baz"], [3, "qux"]]
```

使用 ES6 的解构可以非常容易地在循环中拆分键/值对：

```
const a = ["foo", "bar", "baz", "qux"];

for (const [idx, element] of a.entries()) {
  alert(idx);
  alert(element);
}
// 0
// foo
// 1
// bar
// 2
// baz
// 3
// qux
```

> **注意** 虽然这些方法是 ES6 规范定义的，但在 2017 年底的时候仍有浏览器没有实现它们。

6.2.6 复制和填充方法

ES6 新增了两个方法：批量复制方法 copyWithin()，以及填充数组方法 fill()。这两个方法的函数签名类似，都需要指定既有数组实例上的一个范围，包含开始索引，不包含结束索引。使用这个方法不会改变数组的大小。

使用 fill()方法可以向一个已有的数组中插入全部或部分相同的值。开始索引用于指定开始填充的位置，它是可选的。如果不提供结束索引，则一直填充到数组末尾。负值索引从数组末尾开始计算。也可以将负索引想象成数组长度加上它得到的一个正索引：

```
const zeroes = [0, 0, 0, 0, 0];

// 用 5 填充整个数组
zeroes.fill(5);
console.log(zeroes);  // [5, 5, 5, 5, 5]
zeroes.fill(0);       // 重置

// 用 6 填充索引大于等于 3 的元素
zeroes.fill(6, 3);
console.log(zeroes);  // [0, 0, 0, 6, 6]
zeroes.fill(0);       // 重置

// 用 7 填充索引大于等于 1 且小于 3 的元素
zeroes.fill(7, 1, 3);
console.log(zeroes);  // [0, 7, 7, 0, 0];
zeroes.fill(0);       // 重置

// 用 8 填充索引大于等于 1 且小于 4 的元素
// (-4 + zeroes.length = 1)
// (-1 + zeroes.length = 4)
zeroes.fill(8, -4, -1);
console.log(zeroes);  // [0, 8, 8, 8, 0];
```

`fill()`静默忽略超出数组边界、零长度及方向相反的索引范围：

```
const zeroes = [0, 0, 0, 0, 0];

// 索引过低，忽略
zeroes.fill(1, -10, -6);
console.log(zeroes);  // [0, 0, 0, 0, 0]

// 索引过高，忽略
zeroes.fill(1, 10, 15);
console.log(zeroes);  // [0, 0, 0, 0, 0]

// 索引反向，忽略
zeroes.fill(2, 4, 2);
console.log(zeroes);  // [0, 0, 0, 0, 0]

// 索引部分可用，填充可用部分
zeroes.fill(4, 3, 10)
console.log(zeroes);  // [0, 0, 0, 4, 4]
```

与 `fill()`不同，`copyWithin()`会按照指定范围浅复制数组中的部分内容，然后将它们插入到指定索引开始的位置。开始索引和结束索引则与 `fill()`使用同样的计算方法：

```
let ints,
    reset = () => ints = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9];
reset();

// 从 ints 中复制索引 0 开始的内容，插入到索引 5 开始的位置
// 在源索引或目标索引到达数组边界时停止
ints.copyWithin(5);
console.log(ints);  // [0, 1, 2, 3, 4, 0, 1, 2, 3, 4]
reset();

// 从 ints 中复制索引 5 开始的内容，插入到索引 0 开始的位置
ints.copyWithin(0, 5);
console.log(ints);  // [5, 6, 7, 8, 9, 5, 6, 7, 8, 9]
```

```
reset();

// 从 ints 中复制索引 0 开始到索引 3 结束的内容
// 插入到索引 4 开始的位置
ints.copyWithin(4, 0, 3);
alert(ints);  // [0, 1, 2, 3, 0, 1, 2, 7, 8, 9]
reset();

// JavaScript 引擎在插值前会完整复制范围内的值
// 因此复制期间不存在重写的风险
ints.copyWithin(2, 0, 6);
alert(ints);  // [0, 1, 0, 1, 2, 3, 4, 5, 8, 9]
reset();

// 支持负索引值，与 fill()相对于数组末尾计算正向索引的过程是一样的
ints.copyWithin(-4, -7, -3);
alert(ints);  // [0, 1, 2, 3, 4, 5, 3, 4, 5, 6]
```

copyWithin()静默忽略超出数组边界、零长度及方向相反的索引范围：

```
let ints,
    reset = () => ints = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9];
reset();

// 索引过低，忽略
ints.copyWithin(1, -15, -12);
alert(ints);  // [0, 1, 2, 3, 4, 5, 6, 7, 8, 9];
reset()

// 索引过高，忽略
ints.copyWithin(1, 12, 15);
alert(ints);  // [0, 1, 2, 3, 4, 5, 6, 7, 8, 9];
reset();

// 索引反向，忽略
ints.copyWithin(2, 4, 2);
alert(ints);  // [0, 1, 2, 3, 4, 5, 6, 7, 8, 9];
reset();

// 索引部分可用，复制、填充可用部分
ints.copyWithin(4, 7, 10)
alert(ints);  // [0, 1, 2, 3, 7, 8, 9, 7, 8, 9];
```

6.2.7 转换方法

前面提到过，所有对象都有toLocaleString()、toString()和valueOf()方法。其中，valueOf()返回的还是数组本身。而 toString()返回由数组中每个值的等效字符串拼接而成的一个逗号分隔的字符串。也就是说，对数组的每个值都会调用其toString()方法，以得到最终的字符串。来看下面的例子：

```
let colors = ["red", "blue", "green"]; // 创建一个包含 3 个字符串的数组
alert(colors.toString());   // red,blue,green
alert(colors.valueOf());    // red,blue,green
alert(colors);              // red,blue,green
```

首先是被显式调用的 toString()和 valueOf()方法，它们分别返回了数组的字符串表示，即将