对象的扩展

1. 属性的简介表示法

如果属性值是一个变量，且变量名和属性名一致，则对象键值对可以缩写成属性名：

```
const foo = 'bar';
const baz = {foo};
// 等同于
const baz = {foo: foo};
baz // {foo: "bar"}
```

除了属性简写，方法也可以简写：

```
const o = {
  method() { return "Hello!"; }
};
// 等同于
const o = {
  method: function() { return "Hello!"; }
};
```

简写的对象方法不能用作构造函数，会报错。

```
const obj = {
  f() {
    this.foo = 'bar';
  }
};
new obj.f() // obj.f is not a constructor

const obj2 = {
  f: function() {
    this.foo = 'bar';
  }
};
new obj2.f() // f {foo: 'bar'}
```

2. 属性名表达式

ES6 允许字面量定义对象时，用表达式作为对象的属性名，即把表达式放在方括号内。

```
let lastWord = 'last word';

const a = {
  'first word': 'hello',
  [lastWord]: 'world' // 用表达式作为对象属性名
};

a['first word'] // "hello"
a[lastWord] // "world"
a['last word'] // "world"
```

属性名表达式与简洁表示法，不能同时使用，会报错。

```
const foo = 'bar';
const bar = 'abc';
const baz = { [foo] }; // Unexpected token '['

const foo = 'bar';
const baz = { [foo]: 'abc'};
baz // {bar: 'abc'}
```

3. 方法的属性名

函数的 name 属性，返回函数名。对象方法也是函数，因此也有 name 属性。

```
const person = {
  sayName() {
    console.log('hello!');
  },
};
person.sayName.name   // "sayName"
```

如果对象的方法使用了取值函数（getter）和存值函数（setter），则 `name` 属性不是在该方法上面，而是该方法的属性的描述对象的 `get` 和 `set` 属性上面，返回值是方法名前加上 `get` 和 `set`。

```
const obj = {
  get foo() {},
  set foo(x) {}
};
obj.foo.name; // TypeError: Cannot read properties of undefined (reading 'name')

const descriptor = Object.getOwnPropertyDescriptor(obj, 'foo');
descriptor.get.name // "get foo"
descriptor.set.name // "set foo"
```

4. super 关键字

this 关键字总是指向函数所在的当前对象，ES6 又新增了另一个类似的关键字 super，指向当前对象的原型对象：

```
const proto = {
  foo: 'hello'
};

const obj = {
  foo: 'world',
  find() {
    return super.foo;
  }
};

Object.setPrototypeOf(obj, proto); // 设置 obj 的原型是 proto
obj.find() // "hello", 相当于 proto.foo
```

super.foo 等同于 Object.getPrototypeOf(this).foo。

super 关键字表示原型对象时，只能用在对象的方法之中，用在其他地方都会报错。

```
 // 下面三种情况均报错: 'super' keyword unexpected here
const obj = { foo: super.foo };

const obj = { foo: () => super.foo };

const obj = {
  foo: function () {
    return super.foo
  }
}
```

5. 对象的扩展运算符

5.1. 解构赋值

对象的解构赋值用于从一个对象取值，相当于将目标对象自身的所有可遍历的（enumerable）、但尚未被读取的属性，分配到指定的对象上面。所有的键和它们的值，都会拷贝到新对象上面。

```
let { x, ...z } = { x: 1, a: 3, b: 4 };
x // 1
z // { a: 3, b: 4 }
```

由于解构赋值要求等号右边是一个对象，所以如果等号右边是 `undefined` 或 `null`，就会报错，因为它们无法转为对象。

```
let { ...z } = null; // 运行时错误
let { ...z } = undefined; // 运行时错误
```

解构赋值必须是最后一个参数，否则会报错。

```
let { ...x, y } = { a: 1, b: 2, y: 3 }; // Rest element must be last element
let { x, ...y, ...z } = { x: 1, a: 1, b: 2 };; // Rest element must be last
element
```

解构赋值的拷贝是浅拷贝，即如果一个键的值是复合类型的值（数组、对象、函数）、那么解构赋值拷贝的是这个值的引用，而不是这个值的副本。

```
let obj = { a: { b: 1 } };
let { ...x } = obj;
obj.a.b = 2;
x.a.b // 2
```

5.2. 扩展运算符

对象的扩展运算符（`...`）用于取出参数对象的所有可遍历属性，拷贝到当前对象之中。

```
let z = { a: 3, b: 4 };
let n = { ...z };
n // { a: 3, b: 4 }
```

由于数组是特殊的对象，所以对象的扩展运算符也可以用于数组。

```
let foo = { ...['a', 'b', 'c'] };
foo // {0: "a", 1: "b", 2: "c"}
```

```
// 等同于 {...Object(true)}
{...true} // {}

// 等同于 {...Object(undefined)}
{...undefined} // {}

// 等同于 {...Object(null)}
{...null} // {}
```

如果扩展运算符后面是字符串，它会自动转成一个类似数组的对象，因此返回的不是空对象。

```
{...'hello'} // {0: "h", 1: "e", 2: "l", 3: "l", 4: "o"}
```

对象的扩展运算符，只会返回参数对象自身的、可枚举的属性，尤其是用于类的实例对象时。

```
class C {
  p = 12;
  m() {}
}

let c = new C();
let clone = { ...c };

clone.p; // 12
clone.m(); // clone.m is not a function
```

不会返回 `c` 的方法 `c.m()`，因为这个方法定义在 `C` 的原型对象上。

对象的扩展运算符等同于使用 `Object.assign()` 方法。

```
let aClone = { ...a };
// 等同于
let aClone = Object.assign({}, a);
```

上面的例子只是拷贝了对象实例的属性。

扩展运算符可以用于合并两个对象。

```
let ab = { ...a, ...b };
// 等同于
let ab = Object.assign({}, a, b);
```

如果用户自定义的属性，放在扩展运算符后面，则扩展运算符内部的同名属性会被覆盖掉。

```
let aWithOverrides = { ...a, x: 1, y: 2 };
// 等同于
let aWithOverrides = { ...a, ...{ x: 1, y: 2 } };
// 等同于
let x = 1, y = 2, aWithOverrides = { ...a, x, y };
// 等同于
let aWithOverrides = Object.assign({}, a, { x: 1, y: 2 });
```

这用来修改现有对象部分的属性就很方便了。

```
let newVersion = {
  ...previousVersion,
  name: 'New Name' // Override the name property
};
```