**第9章 对象的扩展**

**9.1 属性的简洁表示法**

ES6允许直接写入变量和函数作为对象的属性和方法。这样的书写更加简洁。

1. var foo = 'bar';

2. var baz = {foo};

3.

4. // 等于

5. var baz = {foo: foo};

6.

上面的代码表面，ES6允许在对象中只写属性名，不写属性值。这时，属性值等于属性名所代表的变量。下面是另一个例子。

1. function f(x, y) {

2.     return {x, y};

3. }

4.

5. // 等同于

6. function f(x, y) {

7.     return {

8.         x: x,

9.         y: y

10.     };

11. }

12.

13. f(1, 2); // Object {x: 1, y: 2}

14.

除了属性可以简写，方法也可以简写。

1. var o = {

2.     method() {

3.         return "Hello world!";

4.     }

5. };

6.

7. // 等同于

8. var o = {

9.     method: function () {

10.         return "Hello world!";

11.     }

12. };

13.

下面是一个实际的例子。

1. var person = {

2.     name: "张山",

3.     birth, // 等同于 birth: birth

4.     // 等同于 hello: function () {...}

5.     hello() {

6.         return console.log("我的名字是", this.name);

7.     }

8. }

9.

这种写法用于函数的返回值会非常方便。

1. function getPoint() {

2.     var x = 1;

3.     var y = 2;

4.     return {x, y};

5. }

6.

7. getPoint(); // {x: 1, y: 2}

8.

CommanJS模块输出变量就非常适合使用简洁写法。

1. var ms = {};

2.

3. function getItem(key) {

4.     return key in ms ? ms[key] : null;

5. }

6.

7. function setItem(key, value) {

8.     ms[key] = value;

9. }

10.

11. function clear() {

12.     ms = {};

13. }

14.

15. module.exports = {getItem, setItem, clear};

16.

17. // 等同于

18. module.exports = {

19.     getItem: getItem,

20.     setItem: setItem,

21.     clear: clear

22. };

23.

24.

属性的赋值器（setter）和取值器（getter）事实上也采用了这种写法。

1. var cart = {

2.     \_wheels: 4,

3.

4.     get wheels() {

5.         return this.\_wheels;

6.     },

7.

8.     set wheels(value) {

9.         if (value < this.\_wheels) {

10.             throw new Error("数值太小了！");

11.         }

12.         this.\_wheels = value;

13.     }

14. }

15.

注意，简洁写法中属性名总是字符串，这会导致一些看上去比较奇怪的结果。

1. var obj = {

2.     class() {}

3. };

4.

5. // 等同于

6. var obj = {

7.     'class': function () {}

8. };

9.

上面的代码中，class是字符串，所以不会因为它属于关键字而导致语法解析报错。

如果某个方法的值是一个Generator函数，则其前面需要加上星号。

1. var obj = {

2.     \* m() {

3.         yield "hello world";

4.     }

5. };

6.

**9.2 属性名表达式**

JavaScript语言定义对象的属性有两种方法。

1. // 方法一

2. Obj.foo = true;

3.

4. // 方法二

5. obj["a" + "bc"] = 123;

6.

上面的方法一是直接用标识符作为属性名；方法二是用表达式作为属性名，这是要将表达式放在方括号内。

但是，如果使用字面量方式定义对象（使用大括号），在ES5中只能使用方法一（标识符）定义属性。

1. var obj = {

2.     foo: true,

3.     abc: 123

4. };

5.

ES6允许字面量定义对象时用方法二（表达式作为对象的属性名），即把表达式放在方括号内。

1. let propKey = 'foo';

2.

3. let obj = {

4.     [propKey]: true,

5.     ['a'+ 'bc']: 123

6. };

7.

下面是另一个例子。

1. var lastWork = "last word";

2.

3. var a = {

4.     'first word': 'hello',

5.     [lastWork]: 'word'

6. };

7.

8. a['first word']; // hello

9. a[lastWork]; // word

10. a['last word']; // word

11.

表达式还可以用于定义方法名。

1. let obj = {

2.     ['a' + 'bc']() {

3.         return 'hi!';

4.     }

5. };

6.

注意，属性名表达式与简洁表示法不能同时使用，否则会报错。

1. // 报错

2. var foo = 'bar';

3. var bar = 'abc';

4. var baz = { [foo] };

5.

6. // 正确

7. var foo = 'bar';

8. var baz = { [foo]: 'abc'};

9.

**9.3 方法的name属性**

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。

1. var person = {

2.     sayName: function () {

3.         return console.log(this.name);

4.     },

5.

6.     get firstName() {

7.         return "Nicholas";

8.     }

9. };

10.

11. person.sayName.name; // sayName

12. person.firstName.name; // get firstName

13.

上面的代码中，方法的name属性返回函数名（即方法名）。如果使用了取值函数，则会在方法名前加上get。如果是存值函数，方法名的前面会加上set。

有两种特殊情况：bind方法创造的函数，name属性返回“bound”加上原函数名；Function构造函数创造的函数，name属性返回“anonymous”。

1. (new Function()).name; // "anonymous"

2.

3. var doSomething = function () {

4.     // ...

5. };

6.

7. doSomething.name; // "bound doSomething"

8.

如果对象的方法是一个Symbol值，那么name属性返回的是这个Symbol值的描述。

1. const key1 = Symbol("description");

2. const key2 = Symbol();

3.

4. let obj = {

5.     [key1]() {},

6.     [key2]() {}

7. };

8.

9. obj[key1].name; // "description"

10. obj[key2].name; // ""

11.

上面的代码中，key1对应的Symbol值有描述，key2没有。

**9.4 Object.is()**

Object.is用来比较两个值是否严格相等。它与严格比较运算符（===）的行为基本一致。

1. Object.is('foo', 'foo');

2. // true

3.

4. Object.is({}, {});

5. // false

6.

不同之处只有两个：一是+0不等于-0，二是NaN等于自身。

1. +0 === -0; // true

2.

3. NaN === NaN; // false

4.

5. Object.is(+0, -0); //false

6. Object.is(NaN, NaN);

7.

ES5可以通过下面的代码部署Object.is。

1. Object.defineProperty(Object, "is", {

2.     value: function (x, y) {

3.         if (x === y) {

4.             // 针对+0不等于-0的情况

5.             return x !== 0 || 1 / x === 1 / y;

6.         }

7.         // 针对NaN的情况

8.         return x !== x && y !== y;

9.     },

10.     configurable: true,

11.     enumerable: false,

12.     writable: true

13. });

14.

**9.5 Object.assign()**

Object.assign方法用来将源对象（source）的所有可枚举属性复制到目标对象（target）。它至少需要两个对象最为参数，第一个参数是目标对象，后面的参数都是源对象。只要有一个参数不是对象，就会抛出TypeError错误。

1. var target = { a: 1 };

2. var source1 = { b: 2 };

3. var source2 = { c: 3 };

4.

5. Object.assign(target, source1, source2);

6.

7. target // { a: 1, b: 2, c: 3}

8.

注意，如果目标对象与源对象有同名属性，或多个源对象有同名属性，则后面的属性会覆盖前面的属性。

1. var target = { a: 1, b: 1 };

2. var source1 = { b: 2, c: 2 };

3. var source2 = { c: 3 };

4.

5. Object.assign(target, source1, source2);

6.

7. target // { a: 1, b: 2, c: 3}

8.

Object.assign只复制自身属性，不可枚举的属性（enumerable为false）和继承的属性不会被复制。

1. Object.assign({b: 'c'}, Object.defineProperty({}, "invisible", {

2.     enumerable: false,

3.     value: "hello"

4. }));

5.

6. // {b: 'c'}

7.

上面的代码中，Object.assign要复制的对象只有一个不可枚举属性invisible，这个属性并没有被复制进去。

属性名为Symbol值得属性，也会被Object.assign复制。

1. Object.assign({a: "b"}, {[Symbol('c')]: d});

2.

3. // {a: 'b', Symbol('c'): d}

4.

对于嵌套的对象，Object.assign的处理方法是替换，而不是添加。

1. var target = {a: {b: 'c', d: 'e'}};

2. var source = {a: {b: "hello"}};

3.

4. Object.assign(target, source);

5. // {a: {b: 'hello'}}

6.

上面的代码，target对象的a属性被source对象的a属性整个替换掉了，不会得到**{a: {b: 'hello', d: 'e'}}**的结果。这通常不是开发者想要的，需要特别小心。有一些函数库提供Object.assign的定制版本（比如Lodash的\_.defaultsDeep方法），可以解决深复制的问题。

注意，Object.assign可以用于数组，但是会将其是为对象。

1. Object.assign([1,2,3], [4,5]);

2. // 4,5,3

3.

上面的代码中，Object.assign把数组视为属性名为0，1，2的对象，因此目标数组的0号属性4覆盖了原数组的0号属性1。

Object.assign方法有很多用处。

**为对象添加属性**

1. Object.assign(SomeClass.prototype, {

2.     someMethod(arg1, arg2) {

3.         // ...

4.     },

5.     anotherMethod() {

6.         // ...

7.     }

8. });

9.

10. // 等同于下面的写法

11. SomeClass.prototype.someMethod = function (arg1, arg2) {

12.     // ...

13. };

14. SomeClass.prototype.anotherMethod = function () {

15.     // ...

16. };

17.

上面的代码使用了对象属性的简洁表示法，直接将两个函数放在大括号中，再使用assign方法添加到SomeClass.prototype中。

**克隆对象**

1. function clone(origin) {

2.     return Object.assign({}, origin);

3. }

4.

上面的代码将原始对象复制到一个空对象，就得到了原始对象的克隆。

不过，采用这种方法克隆，只能克隆原始对象自身的值，不能克隆它继承的值。如果想要保持继承链，可以采用下面的代码。

1. function clone(origin) {

2.     let originProto = Object.getPrototypeOf(origin);

3.     return Object.assign(Object.create(originProto), origin);

4. }

5.

**合并多个对象**

将多个对象合并到某个对象。

1. const merge = (target, ...source) => {Object.assign(target, ...source);}

2.

3.

如果希望合并后返回一个新对象，可以改写上面的函数，对一个空对象合并。

1. const merge = (...source) => {Object.assign({}, ...source);}

2.

**为属性指定默认值**

1. const  DEFAULTS = {

2.     logLevel: 0,

3.     outputFormat: "html"

4. };

5.

6. function processContent(options) {

7.     let options = Object.assign({}, DEFAULTS, options);

8. }

9.

上面的代码中，DEFAULTS对象是默认值，options对象是用户提供的参数。Object.assign方法将DEFAULTS和options合并成一个新对象，如果两者有同名属性，则options的属性值会覆盖DEFAULTS的属性值。

注意，由于存在深复制的问题，DEFAULTS对象和options对象的所有属性的值都只能是简单类型，而不能指向另一个对象。否则，将导致DEFAULTS对象的该属性不起作用。

**9.6 属性的可枚举性**

对象的每个属性都有一个描述对象（Descriptor），用于控制该属性的行为，Object.getOwnPropertyDescriptor方法可以获取该属性的描述对象。

1. let obj = {foo: 123};

2. var descriptor = Object.getOwnPropertyDescriptor(obj, 'foo');

3.

4. // { value: 123, writable: true, enumerable: true, configurable: true }

5.

描述对象的enmuerable属性称为“可枚举性”，如果该属性为false，就表示某些操作会忽略当前属性。

ES5有三个操作会忽略enumerable为false的属性。

* for…in循环：只遍历自身的和继承的可枚举属性。
* Object.keys()：返回对象自身的所有可枚举属性的键名。
* JSON.stringify()：只串行化对象自身的可枚举属性。

ES6新增了两个操作，会忽略enumerable为false的属性。

* Object.assign()：只复制对象自身的可枚举属性。
* Reflect.enumrate()：返回所有for…in循环会遍历的属性。

这5个操作中，只有for…in和Reflect.enumrate()会返回继承的属性。实际上，引入enumerable的最初目的，就是让某些属性可以规避掉for…in操作。比如，对象原型的toString方法，以及数组的length属性，就通过这种手段而不会被for…in遍历到。

1. Object.getOwnPropertyDescriptor(Object.prototype, "toString").enumerable

2. // false

1. Object.getOwnPropertyDescriptor([], "length").enumerable

2. // false

另外，ES6规定，所有class的原型的方法都是不可枚举的。

1. Object.getOwnPropertyDescriptor(class {foo() {}}.prototype, "foo").enumerable

2. // false

总的来书，操作中引入继承的属性会让问题复杂化，大多数时候，我们只关心对象自身的属性。所以，尽量不要用for…in循环，而用Object.keys()代替。

**9.7 属性的遍历**

for…in

for…in循环遍历对象自身的和继承的可枚举属性（不包含Symbol属性）。

Object.keys(obj)

Object.keys返回一个数组，包括对象自身（不含继承的）所有可枚举属性（不含Symbol属性）。

Object.getOwnPropertyNames(obj)

Object.getOwnPropertyNames返回一个数组，包含对象自身的所有属性（不含Symbol属性，但是包括不可枚举属性）。

Object.getOwnPropertySymbols(obj)

Object.getOwnPropertySymbols返回一个数组，包含自身的所有Symbol属性。

Reflect.ownKeys(obj)

Reflect.ownKeys返回一个数组，包括对象自身的所有属性，不关属性名是Symbol或字符串，也不管是否可枚举。

Reflect.enumerate(obj)

Reflect.enumerate返回一个Iterator对象，遍历对象自身的继承的所有可枚举属性（不含Symbol属性），与for…in循环相同。

以上6种方法遍历对象的属性遵守同样的属性遍历次序规则。

* 首先遍历所有属性名为数值的属性，按照数字排序。
* 其次遍历所有属性名为字符串的属性，按照生成时间排序。
* 最后遍历所有属性名为Symbol值的属性，按照生成时间排序。

1. Reflect.ownKeys({[Symbol()]: 0, b: 0, 10: 0, 2: 0, a: 0})

2. // [ '2', '10', 'b', 'a', Symbol() ]

上面的代码种，Reflect.ownKeys方法返回一个数组，包含了参数对象的所有属性。这个数组的属性次序是这样的，首先是数值属性2和10，其次是字符串属性b和a，最后是Symbol属性。

**9.8 \_\_proto\_\_属性，Object.setPrototypeOf()，Object.getPrototypeOf()**

**\_\_proto\_\_属性**

\_\_proto\_\_属性（前后各两个下划线）用来读取或设置当前对象的prototype对象。目前，所有浏览器（包括IE11）都不部署了这个属性。

// ES6的写法

1. var obj = {

2.     method: function() {

3.         // ....

4.     }

5. }

6.

7. obj.\_\_proto\_\_ = someOtherObj;

8.

9. // ES5的写法

10. var obj = Object.create(someOtherObj);

11.

12. obj.method = function () {

13.     // ...

14. }

15.

该属性没有写入ES6的正文，而是写入了附录，原因是\_\_proto\_\_前后的双下划线说明了它本质上是一个内部属性，而不是一个正式的对外的API，只是由于浏览器广泛支持，才被加入了ES6。标准明确规定，只有浏览器必须要部署这个属性，而其他运行环境不一定要部署，而且新的代码最好认为这个属性是不存在的。因此，无论从语义的角度，还是从兼容性的角度，都不要使用这个属性，而是使用Object.setPrototypeOf()（写操作）、Object.getPrototypeOf()（读操作）或Object.create()（生成操作）代替。

在实现上，\_\_proto\_\_调用的是Object.prototype.\_\_proto\_\_，具体实现如下。

1. Object.defineProperty(Object.prototype, "\_\_proto\_\_", {

2.     get() {

3.         let \_thisObj = Object(this);

4.         return Object.getPrototypeOf(\_thisObj);

5.     },

6.     set(proto) {

7.         if (this === undefined || this == null) {

8.             throw new TypeError();

9.         }

10.         if(!isObject(this)) {

11.             return undefined;

12.         }

13.         if (!isObject(proto)) {

14.             return undefined;

15.         }

16.         let status = Reflect.setPrototypeOf(this, proto);

17.         if(!status) {

18.             throw new TypeError();

19.         }

20.     }

21. });

22.

23. function isObject(value) {

24.     return Object(value) === value;

25. }

26.

如果一个对象本身部署了\_\_proto\_\_属性，则该属性的值就是对象的原型。

1. Object.getPrototypeOf({ \_\_proto\_\_: null} )

2. // null

3. Object.setPrototypeOf()

Object.setPrototypeOf()方法的作用与\_\_proto\_\_相同，用于设置一个对象的prototype对象。

它是ES6正式推荐的设置原型对象的方法。

1. // 格式

2. Object.getPrototypeOf(object, prototype);

3.

4. // 用法

5. var o = Object.setPrototypeOf({}, null);

6.

7.

该方法等同于下面的函数。

1. function (obj, proto) {

2.     obj.\_\_proto = proto;

3.     return obj;

4. }

5.

小面是一个例子

1. let proto = {};

2. let obj = {x: 10};

3.

4. Object.setPrototypeOf(obj, proto);

5.

6. proto.y = 20;

7. proto.z = 40;

8.

9. obj.x // 10

10. obj.y // 20

11. obj.z // 40

12.

13.

上面的代码将proto对象设置为obj对象的原型，所以从obj对象可以读取proto对象的属性。

1. Object.getPrototypeOf()

2.

该方法与setPrototypeOf()方法配套，读取一个对象的prototype对象。

1. Object.getPrototypeOf(obj);

2.

下面是一个例子。

1. function Rectangle() {}

2.

3. var rec = new Rectangle();

4.

5. Object.getPrototypeOf(rec) === Rectangle.prototype

6. // true

7.

8. Object.setPrototypeOf(rec, Object.prototype);

9. Object.getPrototypeOf(rec) === Rectangle.prototype

10. // false

11.

**9.9 对象的扩展运算符**

目前，ES7有一个提案（https://github.com/sebmarkbage/ecmascript-rest-spread），将rest参数扩展运算符（…）引入对象。Babel转码器已经支持这项功能。

**Rest 参数**

Rest参数用于从一个对象取值，相当于将所有可遍历但尚未被读取的属性，分配到指定的对象上。所有的键及其值都会复制到新对象上。

1. let {x, y, ...z} = { x: 1, y: 2, a: 3, b: 4}

2.

3. // x = 1

4. // y = 2

5. // z = {a:3, b: 4}

6.

上面的代码中，变量z是rest参数所在的对象。它获取等号右边的所有尚未读取的键（a和b），将他们及其值复制过来。

注意，rest参数的复制是浅复制，即如果一个键的值是复合类型的值（数组，对象，函数），那么rest参数复制的是这个值的引用，而不是这个值的副本。

1. let obj = { a: {b: 1} };

2. let {...x} = obj;

3.

4. obj.a.b = 2;

5.

6. x.a.b // 2

7.

上面的代码中，x是rest参数，复制了对象obj的a属性。a属性引用了一个对象，修改这个对象的值会影响到rest参数对它的引用。

另外，rest参数不会复制继承自原型对象的属性。

1. let o1 = {a: 1};

2. let o2 = {b: 2};

3.

4. o2.\_\_proto\_\_ = o1;

5. let o3 = {...o2};

6.

7. o3  // {b: 2}

8.

上面的代码中，对象o3是o2的复制，但是只复制了o2自身的属性，没有复制其原型对象o1的属性。

**扩展运算符**

扩展运算符用于取出参数对象的所有可遍历属性，复制到当前对象中。

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

2. let n = { ...z };

3. n // { a: 3, b: 4 }

4.

这等同于使用Object.assign方法。

1. let aClone = { ...a };

2. // 等同于

3. let aClone = Object.assign({}, a);

4.

扩展运算符可用于合并两个对象。

1. let ab = { …a, …b};

2.

扩展运算符还可以用于自定义属性，会在新对象中覆盖掉原有参数。

1. let aWithOverrides = {...a, x: 1, y: 2};

2. // 等同于

3. let aWithOverrides = { ...a, ...{x: 1, y: 2}};

4. // 等同于

5. let x = 1, y =2, aWithOverrides = {...a, x, y}

6. // 等同于

7. let aWithOverrides = Object.assign({}, a, {x: 1, y: 2});

8.

上面的代码中，a对象的x属性和y属性，复制到新对象后会被覆盖掉。

如果把自定义属性放在扩展运算符前面，就变成了设置新对象的默认值。

1. let aWithDefaults = {x: 1, y: 2, ...a};

2. // 等同于

3. let aWtihDefaults = Object.assign({}, {x: 1, y: 2}, a);

4. // 等同于

5. let aWithDefaults = Object.assign({x: 1, y: 2}, a);

6.

扩展运算符的参数对象中，如果有**取值函数get**，那么这个函数是会执行的。

1. // 并不会抛出错误，因为x属性只是被定义，但未执行

2. let aWithXGetter = {

3.   ...a,

4.   get x() {

5.     throw new Error("Not thrown yet");

6.   },

7. };

8.

9. console.log(aWithXGetter);

10.

11. // 会抛出错误，因为x属性被执行了

12. let runtimeError = {

13.   ...a,

14.   ...{

15.     get x() {

16.       throw new Error("thrown now");

17.     },

18.   },

19. };

如果扩展运算符的参数是null或undefined，则会被忽略，不会报错。

1. let emptyObject = {…null, …undefined}; // 不报错

2.