对象

1. 对象引用

如果不同的变量名指向同一个对象，那么它们都是这个对象的引用，也就是说指向同一个内存地址。修改其中一个变量，会影响到其他所有变量。

```
let o1 = {};
let o2 = o1;

o1.a = 1;
o2.a; // 1

o2.b = 2;
o1.b; // 2
```

如果取消某一个变量对于原对象的引用，比如将其变成原始类型值，不会影响到另一个变量。

```
let o1 = {};
let o2 = o1;

// 重新赋值（并非修改属性），会改变引用地址，取消了对于原对象的引用
o1 = 1;
o2; // {}
```

只要对象被整体重新赋值了，那么就不会影响到另一对象。被重新赋值为一个引用类型时，该对象的指向的地址就变了，不会影响其他对象。被重新赋值为一个值类型是，存在栈中，就没有指向了，就也不会影响其他对象了。

这种引用只局限于对象，如果两个变量指向同一个原始类型的值。那么，变量这时都是值的拷贝。

```
let x = 1;
let y = x;

x = 2;
y // 1
```

1.1. 属性读取

数字键可以不加引号，因为会自动转成字符串。

```
let obj = {
  0.7: 'Hello World'
```

```
};
obj['0.7']; // "Hello World"
obj[0.7];   // "Hello World"
```

数值键名不能使用点运算符（因为会被当成小数点），只能使用方括号运算符。

```
let obj = {
  123: 'hello world'
};
obj.123;  // 报错
obj[123]; // "hello world"
```

1.2. 属性删除

删除一个不存在的属性，`delete` 不报错，而且返回 `true`。

```
let obj = {};
delete obj.p; // true
```

1.3. 属性是否存在

1.3.1. `in` 运算符

```
let obj = { p: 1 };
'p' in obj;        // true
'toString' in obj; // true
```

对象 `obj` 本身并没有 `toString` 属性，但是 `in` 运算符会返回 `true`，因为这个属性是继承的。

和 `Reflect.has()` API 相同：

```
let obj = { p: 1 };
Reflect.has(obj, 'p');        // true
Reflect.has(obj, 'toString'); // true
```

1.3.2. Object.prototype.hasOwnProperty()

`hasOwnProperty()` 方法判断是否是对象自身的属性。

```
let obj3 = {a: 1, b: 2};
'a' in obj3;                    // true
'toString' in obj3;             // true
```

```
obj3.hasOwnProperty('toString'); // false

let obj4 = {'c': 4, 'toString': 5}
obj4.hasOwnProperty('toString'); //  true
```

如果对象含有 Symbol 类型的属性，使用 hasOwnProperty 也无法获取到。只能通过 Object.getOwnPropertySymbols() 或 Reflect.ownKeys() 方法：

```
let o = {a: 1, [Symbol('foo')]: 2};
Symbol('foo') in o;              // false
o.hasOwnProperty(Symbol('foo')); // false

Object.getOwnPropertySymbols(o); // [Symbol(foo)]
Reflect.ownKeys(o);              // ['a', Symbol(foo)]
```

1.3.3. Object.hasOwn()

如果指定的对象自身有指定的属性，静态方法 Object.hasOwn() 返回 true。如果属性是继承的或者不存在，该方法返回 false。Object.hasOwn() 旨在取代 Object.prototype.hasOwnProperty()。

```
const o = {
  prop: 'exists',
};
Object.hasOwn(o, 'prop');               // true
Object.hasOwn(o, 'toString');           // false
Object.hasOwn(o, 'undeclaredProperty'); // false
```

Object.hasOwn() 解决了 hasOwnProperty 存在的两个问题：

（1）使用 hasOwnProperty 作为属性名称

JavaScript 并不保护属性名称 hasOwnProperty，具有此名称属性的对象可能会返回不正确的结果。

```
const foo = {
  hasOwnProperty() {
    return false;
  },
  bar: "Here be dragons",
};
foo.hasOwnProperty("bar"); // 该重新实现始终返回 false
Object.hasOwn(foo, "bar"); // true
```

（2）由 Object.create(null) 创建的对象

使用 `Object.create(null)` 创建的对象不从 `Object.prototype` 继承，使得 `hasOwnProperty()` 不可访问。

```
const foo = Object.create(null);
foo.prop = "exists";
foo.hasOwnProperty("prop"); // TypeError: foo.hasOwnProperty is not a function
Object.hasOwn(foo, "prop"); // true
```