对象新增方法

1. Object.is()

`Object.is()` 用来比较两个值是否严格相等，与严格比较运算符（`===`）的行为基本一致，只有两点不同：

> - `+0` 不等于 `-0`
> - `NaN` 等于自身

```
+0 === -0; // true
NaN === NaN; // false

Object.is(+0, -0); // false
Object.is(NaN, NaN); // true
```

2. Object.assign()

`Object.assign()` 方法用于对象的合并，第一个参数是目标对象，后面的参数都是源对象。将源对象（`source`）的所有可枚举属性，复制到目标对象（`target`）：

```
const target = { a: 1 };

const source1 = { b: 2 };
const source2 = { c: 3 };

Object.assign(target, source1, source2);
target; // {a:1, b:2, c:3}
```

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。

```
const target = { a: 1, b: 1 };

const source1 = { b: 2, c: 2 };
const source2 = { c: 3 };

Object.assign(target, source1, source2);
target; // {a:1, b:2, c:3}
```

如果只有一个参数，Object.assign() 会直接返回该参数。

```
const obj = {a: 1};
Object.assign(obj) === obj; // true
```

`Object.assign()` 拷贝的属性是有限制的，只拷贝源对象的自身属性（不拷贝继承属性），也不拷贝不可枚举的属性（`enumerable: false`）。

```
Object.assign({b: 'c'},
  Object.defineProperty({}, 'invisible', {
    enumerable: false,
    value: 'hello'
  })
);
// { b: 'c' }
```

上例中，`Object.assign()` 要拷贝的对象只有一个不可枚举属性 `invisible`，这个属性并没有被拷贝进去。

属性名为 Symbol 值的属性，也会被 `Object.assign()` 拷贝。

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

3. Object.getOwnPropertyDescriptors()

ES5 的 `Object.getOwnPropertyDescriptor()` 方法返回指定对象所有自身属性（非继承属性）的描述对象。

```
const obj = {
  foo: 123,
  get bar() { return 'abc' }
};
Object.getOwnPropertyDescriptors(obj);
// {
//     "foo": {
//         "value": 123,
//         "writable": true,
//         "enumerable": true,
//         "configurable": true
//     },
//     "bar": {
//         "enumerable": true,
//         "configurable": true
//     }
// }
```

上例中，`Object.getOwnPropertyDescriptors()` 方法返回一个对象，所有原对象的属性名都是该对象的属性名，对应的属性值就是该属性的描述对象。

4. __proto__属性，Object.setPrototypeOf()，Object.getPrototypeOf()

4.1. __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原型对象（prototype）。所有浏览器（包括IE11）都部署了这个属性。

```
// es5 的写法
const obj = {
  method: function() { ... }
};
obj.__proto__ = someOtherObj;

// es6 的写法
var obj = Object.create(someOtherObj);
obj.method = function() { ... };
```

__proto__ 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，只有浏览器必须部署这个属性，其他运行环境不一定需要部署，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）、Object.create()（生成操作）代替。

实现上，__proto__ 调用的是 Object.prototype.__proto__，具体实现如下。

```
Object.defineProperty(Object.prototype, '__proto__', {
  get() {
    let _thisObj = Object(this);
    return Object.getPrototypeOf(_thisObj);
  },
  set(proto) {
    if (this === undefined || this === null) {
      throw new TypeError();
    }
    if (!isObject(this)) {
      return undefined;
    }
    if (!isObject(proto)) {
      return undefined;
    }
    let status = Reflect.setPrototypeOf(this, proto);
    if (!status) {
      throw new TypeError();
    }
  },
});

function isObject(value) {
  return Object(value) === value;
}
```

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

```
Object.getPrototypeOf({ __proto__: null }); // null
```

4.2. Object.setPrototypeOf()

`Object.setPrototypeOf()` 方法的作用与 `__proto__` 相同，用来设置一个对象的原型对象（prototype），返回参数对象本身。它是 ES6 正式推荐的设置原型对象的方法。

```
// 格式
Object.setPrototypeOf(object, prototype)

// 用法
const o = Object.setPrototypeOf({}, null);
```

`setPrototypeOf()` 的例子：

```
let proto = {};
let obj = { x: 10 };
Object.setPrototypeOf(obj, proto); // 将 proto 对象设为 obj 对象的原型

proto.y = 20;
proto.z = 40;

// 从 obj 对象可以读取其原型 proto 对象的属性
obj.x // 10
obj.y // 20
obj.z // 40
```

如果第一个参数不是对象，会自动转为对象。但是由于返回的还是第一个参数，所以这个操作不会产生任何效果。

```
Object.setPrototypeOf(1, {}) === 1; // true
Object.setPrototypeOf('foo', {}) === 'foo'; // true
Object.setPrototypeOf(true, {}) === true; // true
```

由于 `undefined` 和 `null` 无法转为对象，所以如果第一个参数是 `undefined` 或 `null`，就会报错。

```
Object.setPrototypeOf(undefined, {});
// TypeError: Object.setPrototypeOf called on null or undefined

Object.setPrototypeOf(null, {});
// TypeError: Object.setPrototypeOf called on null or undefined
```

4.3. Object.getPrototypeOf()

该方法与 `Object.setPrototypeOf` 方法配套，用于读取一个对象的原型对象。

如果参数不是对象，会被自动转为对象。

```
// 等同于 Object.getPrototypeOf(Number(1))
Object.getPrototypeOf(1); // Number {[[PrimitiveValue]]: 0}

// 等同于 Object.getPrototypeOf(String('foo'))
Object.getPrototypeOf('foo'); // String {length: 0, [[PrimitiveValue]]: ""}

// 等同于 Object.getPrototypeOf(Boolean(true))
Object.getPrototypeOf(true); // Boolean {[[PrimitiveValue]]: false}

Object.getPrototypeOf(1) === Number.prototype; // true
Object.getPrototypeOf('foo') === String.prototype; // true
Object.getPrototypeOf(true) === Boolean.prototype; // true
```

如果参数是 `undefined` 或 `null`，它们无法转为对象，所以会报错。

```
Object.getPrototypeOf(null);
// TypeError: Cannot convert undefined or null to object

Object.getPrototypeOf(undefined);
// TypeError: Cannot convert undefined or null to object
```

5. Object.keys()，Object.values()，Object.entries()

5.1. Object.keys()

`Object.keys()` 方法返回一个数组，成员是参数对象自身的（不含继承的）所有可遍历（`enumerable`）属性的键名。

```
var obj = { foo: 'bar', baz: 42 };
Object.keys(obj); // ["foo", "baz"]
```

5.2. Object.values()

`Object.values()` 方法返回一个数组，成员是参数对象自身的（不含继承的）所有可遍历（`enumerable`）属性的键值。

```
const obj = { foo: 'bar', baz: 42 };
Object.values(obj); // ["bar", 42]
```

返回数组的成员顺序，是按照数值大小，从小到大遍历的。

```
const obj = { 100: 'a', 2: 'b', 7: 'c' };
Object.keys(obj); // ["2", "7", "100"]
Object.values(obj); // ["b", "c", "a"]
```

`Object.values()` 只返回对象自身的可遍历属性。

```
const obj = Object.create({}, {p: {value: 42}});
Object.values(obj); // []
```

上例中，`Object.create` 方法的第二个参数添加的对象属性（属性 `p`），如果不显式声明，默认是不可遍历的，因为 `p` 的属性描述对象的 `enumerable` 默认是 `false`，`Object.values` 不会返回这个属性。只要把 `enumerable` 改成 `true`，`Object.values` 就会返回属性p的值。

```
const obj = Object.create({}, {p:
  {
    value: 42,
    enumerable: true
  }
});
Object.values(obj) // [42]
```

`Object.values` 会过滤属性名为 Symbol 值的属性：

```
Object.values({ [Symbol()]: 123, foo: 'abc' }); // ['abc']
```

如果 `Object.values` 方法的参数是一个字符串，会返回各个字符组成的一个数组。

```
Object.values('foo'); // ['f', 'o', 'o']
```

上例中，字符串会先转成一个类似数组的对象。字符串的每个字符，就是该对象的一个属性。因此，`Object.values` 返回每个属性的键值，就是各个字符组成的一个数组。

如果参数不是对象，`Object.values` 会先将其转为对象。由于数值和布尔值的包装对象，都不会为实例添加非继承的属性。所以，`Object.values` 会返回空数组。

```
Object.values(42); // []
Object.values(true); // []
```

5.3. Object.entries()

`Object.entries()` 方法返回一个数组，成员是参数对象自身的（不含继承的）所有可遍历（`enumerable`）属性的键值对数组。

```
const obj = { foo: 'bar', baz: 42 };
Object.entries(obj); // [ ["foo", "bar"], ["baz", 42] ]
```

除了返回值不一样，该方法的行为与 `Object.values` 基本一致。

如果原对象的属性名是一个 Symbol 值，该属性会被忽略。

```
Object.entries({ [Symbol()]: 123, foo: 'abc' }); // [ [ 'foo', 'abc' ] ]
```

`Object.entries()` 的基本用途是遍历对象的属性：

```
let obj = { one: 1, two: 2 };
for (let [k, v] of Object.entries(obj)) {
  console.log(
    `${JSON.stringify(k)}: ${JSON.stringify(v)}`
  );
}
// "one": 1
// "two": 2
```

`Object.entries()` 方法的另一个用处是，将对象转为真正的 `Map` 结构：

```
const obj = { foo: 'bar', baz: 42 };
const map = new Map(Object.entries(obj));
map; // Map { foo: "bar", baz: 42 }
```

6. Object.fromEntries()

`Object.fromEntries()` 方法是 `Object.entries()` 的逆操作，用于将一个键值对数组转为对象。

```
Object.fromEntries([['foo', 'bar'], ['baz', 42]]);
// { foo: "bar", baz: 42 }
```

该方法的主要目的，是将键值对的数据结构还原为对象，因此特别适合将 Map 结构转为对象。

```
// 例一
const entries = new Map([['foo', 'bar'], ['baz', 42]]);
Object.fromEntries(entries); // { foo: "bar", baz: 42 }

// 例二
const map = new Map().set('foo', true).set('bar', false);
Object.fromEntries(map); // { foo: true, bar: false }
```

该方法的一个用处是配合 `URLSearchParams` 对象，将查询字符串转为对象。

```
Object.fromEntries(new URLSearchParams('foo=bar&baz=qux'));
// { foo: "bar", baz: "qux" }
```

7. Object.hasOwn()

JavaScript 对象的属性分成两种：自身的属性和继承的属性。对象实例有一个 `hasOwnProperty()` 方法，可以判断某个属性是否为原生属性。静态方法 `Object.hasOwn()` 可以判断是否为自身的属性。

`Object.hasOwn()` 可以接受两个参数，第一个是所要判断的对象，第二个是属性名。

```
const foo = Object.create({a: 123}); // {a: 123} 变成 foo 的原型
foo.b = 456;

Object.hasOwn(foo, 'a'); // false
Object.hasOwn(foo, 'b'); // true
```

上例中，对象 `foo` 的属性 `a` 是继承属性，属性 `b` 是原生属性。

`Object.hasOwn()` 的一个好处是，对于不继承 `Object.prototype` 的对象不会报错，而 `hasOwnProperty()` 是会报错的。

```
const obj = Object.create(null); // null 作为 obj 的原型，即 obj 没有原型，不继承任何
属性
obj.hasOwnProperty('foo'); // TypeError: obj.hasOwnProperty is not a function
Object.hasOwn(obj, 'foo'); // false
```