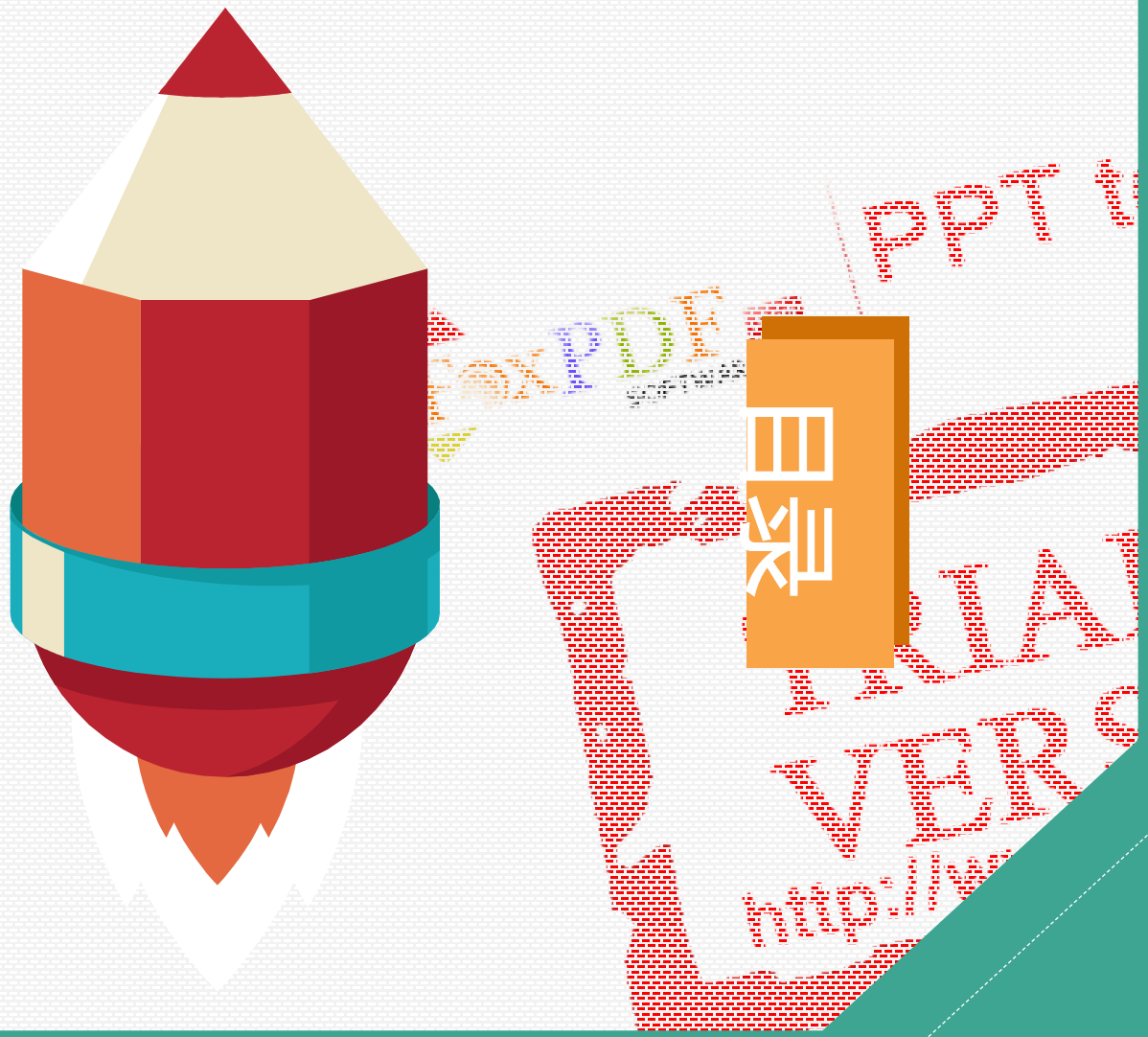
第31课：Symbol方法和对象的方法

主讲老师：万章



symbol数据类型进阶

对象的方法拓展

symbol数据类型进阶

symbol数据类型进阶之for()

如果我们希望重新使用同一个 Symbol 值，Symbol.for方法可以做到这一点。它接受一个字符串作为参数，然后搜索有没有以该参数作为名称的 Symbol 值(这个值必须也得是Symbol.for()创造的)。如果有，就返回这个 Symbol 值，否则就新建并返回一个以该字符串为名称的 Symbol 值。

```
let s1 = Symbol.for('foo');
let s2 = Symbol.for('foo');

s1 === s2 // true
```

```
> let s1 = Symbol('foo');
  let s2 = Symbol.for('foo');

  s1 === s2
< false
>
```

Symbol.for()与Symbol()这两种写法，都会生成新的 Symbol。

它们的区别是：前者会被登记在全局环境中供搜索，后者不会。

Symbol.for()不会每次调用就返回一个新的 Symbol 类型的值，而是会先检查给定的key是否已经存在，如果不存在才会新建一个值。

比如，如果你调用Symbol.for("cat")30 次，每次都会返回同一个 Symbol 值，但是调用Symbol("cat")30 次，会返回 30 个不同的 Symbol 值。

symbol数据类型进阶之keyFor()

Symbol.keyFor方法返回一个已登记的 Symbol 类型值的key

```
> let s1 = Symbol.for("foo");
  let s2 = Symbol.for("foo");

  Symbol.keyFor(s1);
< "foo"
> Symbol.keyFor(s2);
< "foo"
> let s3 = Symbol("foo");
  Symbol.keyFor(s3);
< undefined
>
```

变量s3属于未登记的 Symbol 值，所以返回undefined。

symbol数据类型进阶之内嵌Symbol值

对象的Symbol.hasInstance属性，指向一个内部方法。当其他对象使用instanceof运算符，判断是否为该对象的实例时，会调用这个方法。比如，foo instanceof Foo在语言内部，实际调用的是Foo[Symbol.hasInstance](foo)。

```
> const Even = {
    [Symbol.hasInstance](obj) {
      return Number(obj) % 2 === 0;
    }
  };
< undefined
> 1 instanceof Even
< false
> 2 instanceof Even
< true
>
```

这个方法可以作为对象方法的拓展，或是对于对象继承的逻辑判断进行自定义修饰

symbol数据类型进阶之内嵌Symbol值

对象的Symbol.isConcatSpreadable属性等于一个布尔值，表示该对象用于Array.prototype.concat()时，是否可以展开。

```
let arr1 = ['c', 'd'];
['a', 'b'].concat(arr1, 'e') // ['a', 'b', 'c', 'd', 'e']
arr1[Symbol.isConcatSpreadable] // undefined

let arr2 = ['c', 'd'];
arr2[Symbol.isConcatSpreadable] = false;
['a', 'b'].concat(arr2, 'e') // ['a', 'b', ['c','d'], 'e']
```

```
let obj = {length: 2, 0: 'c', 1: 'd'};
['a', 'b'].concat(obj, 'e') // ['a', 'b', obj, 'e']

obj[Symbol.isConcatSpreadable] = true;
['a', 'b'].concat(obj, 'e') // ['a', 'b', 'c', 'd', 'e']
```

上面代码说明，数组的默认行为是可以展开，Symbol.isConcatSpreadable默认等于undefined。该属性等于true时，也有展开的效果。

类似数组的对象正好相反，默认不展开。它的Symbol.isConcatSpreadable属性设为true，才可以展开。

对象的方法拓展

对象的方法拓展之对象的创造

```
Object.create(proto, [propertiesObject])
```

proto

新创建对象的原型对象。

propertiesObject

可选。如果没有指定为 undefined，则是要添加到新创建对象的可枚举属性（即其自身定义的属性，而不是其原型链上的枚举属性）对象的属性描述符以及相应的属性名称。

一个新对象，带着指定的原型对象和属性。

如果propertiesObject参数是 null ，则抛出一个 TypeError 异常。

对象的方法拓展之Object.is()

ES5 比较两个值是否相等，只有两个运算符：相等运算符（==）和严格相等运算符（===）
它们都有缺点，前者会自动转换数据类型，后者的NaN不等于自身，以及+0等于-0

ES6 提出“Same-value equality”（同值相等）算法，用来解决这个问题。Object.is就是部署这个算法的新方法。它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。

```
Object.is(值1, 值2)
```

不同之处只有两个：一是+0不等于-0，二是NaN等于自身。

对象的方法拓展之Object.assign()

Object.assign方法用于对象的合并，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, b: 1 };

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

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

Object.assign方法的第一个参数是目标对象，后面的参数都是源对象。

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。

对象的方法拓展之Object.assign()

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

如果只有一个参数，Object.assign会直接返回该参数。

```
> Object.assign(2)
< ▼Number {2}
    ▶__proto__: Number
     [[PrimitiveValue]]: 2
```

如果该参数不是对象，则会先转成对象，然后返回。

```
Object.assign(undefined) // 报错
Object.assign(null) // 报错
```

由于undefined和null无法转成对象，所以如果它们作为目标对象参数，就会报错。

如果非对象参数出现在源对象的位置（即非首参数），那么处理规则有所不同。首先，这些参数都会转成对象，如果无法转成对象，就会跳过。这意味着，如果undefined和null不在目标对象参数，就不会报错。

```
let obj = {a: 1};
Object.assign(obj, undefined) === obj // true
Object.assign(obj, null) === obj // true
```

对象的方法拓展之Object.assign()

其他类型的值（即数值、字符串和布尔值）不在首参数，也不会报错。但是，除了字符串会以数组形式，拷贝入目标对象，其他值都不会产生效果。

```
> const v1 = 'abc';
  const v2 = true;
  const v3 = 10;

  const obj = Object.assign({}, v1, v2, v3);
  console.log(obj);
  ▼{0: "a", 1: "b", 2: "c"}
     0: "a"
     1: "b"
     2: "c"
    ▶__proto__: Object
  undefined
>
```

```
> Object(true)
  ▼Boolean {true}
    ▶__proto__: Boolean
     [[PrimitiveValue]]: true
> Object(10)
  ▼Number {10}
    ▶__proto__: Number
     [[PrimitiveValue]]: 10
> Object('abc')
  ▼String {"abc"}
     0: "a"
     1: "b"
     2: "c"
     length: 3
    ▶__proto__: String
     [[PrimitiveValue]]: "abc"
>
```

上面代码中，v1、v2、v3分别是字符串、布尔值和数值，结果只有字符串合入目标对象（以字符数组的形式），数值和布尔值都会被忽略。这是因为只有字符串的包装对象，会产生可枚举属性;对象的内部属性[[PrimitiveValue]]上面，这个属性是不会被Object.assign拷贝的

对象的方法拓展之Object.assign()

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' }
```

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' }
```

对象的方法拓展之Object.assign()的几个核心注意点

```
const obj1 = {a: {b: 1}};
const obj2 = Object.assign({}, obj1);

obj1.a.b = 2;
obj2.a.b // 2
```

（1）浅拷贝

Object.assign方法实行的是浅拷贝，而不是深拷贝。也就是说，如果源对象某个属性的值是对象，那么目标对象拷贝得到的是这个对象的引用。

```
const target = { a: { b: 'c', d: 'e' } }
const source = { a: { b: 'hello' } }
Object.assign(target, source)
// { a: { b: 'hello' } }
```

（2）同名属性的替换

对于这种嵌套的对象，一旦遇到同名属性，Object.assign的处理方法是替换，而不是添加。

对象的方法拓展之Object.assign()的几个核心注意点

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

（3）数组的处理

Object.assign可以用来处理数组，但是会把数组视为对象。

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

```
const source = {
  get foo() { return 1 }
};
const target = {};

Object.assign(target, source)
// { foo: 1 }
```

（4）取值函数的处理

Object.assign只能进行值的复制，如果要复制的值是一个取值函数，那么将求值后再复制。

source对象的foo属性是一个取值函数，Object.assign不会复制这个取值函数，只会拿到值以后，将这个值复制过去。

对象的方法拓展之Object.assign()的常见使用领域

```
> let o={
      constructor(x,y){
          Object.assign(this, {x, y});
      }
  }
< undefined
> o.constructor(1,3)
< undefined
> o
< ▼{constructor: ƒ, x: 1, y: 3}
    ▶constructor: ƒ constructor(x,y)
     x: 1
     y: 3
    ▶__proto__: Object
>
```

（1）为对象添加属性

上面方法通过Object.assign方法，将x属性和y属性添加到Point类的对象实例。

```
Object.assign(SomeClass.prototype, {
  someMethod(arg1, arg2) {
    ...
  },
  anotherMethod() {
    ...
  }
});

// 等同于下面的写法
SomeClass.prototype.someMethod = function (arg1, arg2) {
  ...
};
SomeClass.prototype.anotherMethod = function () {
  ...
};
```

（2）为对象添加方法

这种方法可以避免发生原型被覆盖的问题，我们可以直接把一个对象结构的数据给原型设置上去且不会覆盖原有的，也不用重新定义构造函数

对象的方法拓展之Object.assign()的常见使用领域

```
function clone(origin) {
  return Object.assign({}, origin);
}
```

（3）克隆对象

不过还是那个老问题，这个克隆是浅复制

当被复制的对象内部属性的值是基础类型倒是没啥,如果也是一个对象的话，那么这个克隆知识克隆了该值的内存地址

而且采用这种方法克隆，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。

```
function clone(origin) {
  let originProto = Object.getPrototypeOf(origin);
  return Object.assign(Object.create(originProto), origin);
}
```

```
> function clone(origin) {
      return Object.assign({}, origin);
  }

  let o = {
      x: 1,
      y: 2,
      z:{
          a:1,
          b:2
      }
  };
  let o1 = clone(o);
< undefined
> o.x=999;
  o.z.a=9527
< 9527
> o1
< ▼{x: 1, y: 2, z: {…}}
    x: 1
    y: 2
  ▼z:
      a: 9527
      b: 2
    ▶__proto__: Object
  ▶__proto__: Object
>
```

对象的方法拓展之Object.assign()的常见使用领域

（4）合并多个对象

将多个对象合并到某个对象。

```
const merge =
  (target, ...sources) => Object.assign(target, ...sources);
```

如果希望合并后返回一个新对象，可以改写上面函数，对一个空对象合并。

```
const merge =
  (...sources) => Object.assign({}, ...sources);
```

（5）为属性指定默认值

```
const DEFAULTS = {
  logLevel: 0,
  outputFormat: 'html'
};

function processContent(options) {
  options = Object.assign({}, DEFAULTS, options);
  console.log(options);
  // ...
}
```

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

对象的方法拓展之属性描述对象的获取

ES5 的Object.getOwnPropertyDescriptor()方法会返回某个对象属性的描述对象（descriptor）。ES2017 引入了Object.getOwnPropertyDescriptors()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: {…}, bar: {…}}
    ▼bar:
       configurable: true
       enumerable: true
     ▼get: ƒ bar()
         arguments: (...)
         caller: (...)
         length: 0
         name: "get bar"
       ▶__proto__: ƒ ()
         [[FunctionLocation]]: VM295:3
       ▶[[Scopes]]: Scopes[2]
       set: undefined
     ▶__proto__: Object
    ▼foo:
       configurable: true
       enumerable: true
       value: 123
       writable: true
     ▶__proto__: Object
    ▶__proto__: Object
>
```

该方法的引入目的，主要是为了解决Object.assign()无法正确拷贝get属性和set属性的问题。

```
const source = {
  set foo(value) {
    console.log(value);
  }
};

const target1 = {};
Object.assign(target1, source);

Object.getOwnPropertyDescriptor(target1, 'foo')
// { value: undefined,
//   writable: true,
//   enumerable: true,
//   configurable: true }
```

对象的方法拓展之原型的设置与获取

Object.setPrototypeOf方法的用来设置一个对象的prototype对象，**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);
```

该方法等同于=

```
function setPrototypeOf(obj, proto) {
  obj.__proto__ = proto;
  return obj;
}
```

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
```

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

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

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。

对象的方法拓展之原型的设置与获取

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

```
function Rectangle() {
  // ...
}

const rec = new Rectangle();

Object.getPrototypeOf(rec) === Rectangle.prototype
// true

Object.setPrototypeOf(rec, Object.prototype);
Object.getPrototypeOf(rec) === Rectangle.prototype
// false
```

```
// 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
```

标准示例

如果参数不是对象，会被自动转为对象

如果参数是undefined或null，它们无法转为对象，所

对象的方法拓展之属性名称的遍历

ES5 引入了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"]
```

对象的方法拓展之属性值的遍历

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。Object.values会过滤属性名为 Symbol 值的属性。

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

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

Object.values只返回对象自身的可遍历属性。

上面代码中，属性名为数值的属性，是按照数值大小，从小到大遍历的，因此返回的顺序是b、c、a。

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。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('foo')
// ['f', 'o', 'o']
```

对象的方法拓展之名值对的遍历

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。如果原对象的属性名是一个 Symbol 值，该属性会被忽略。

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

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

```
> let obj = { one: 1, two: 2 };
  for (let [k, v] of Object.entries(obj)) {
    console.log(
      `${k}: ${v}`
    );
  }
  one: 1                                    VM505:3
  two: 2                                    VM505:3
< undefined
>
```

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

Object.entries的基本用途是遍历对象的属性。

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。