ES6 PPT1606171: Object/Array扩展

Thursday, November 5, 2015 10:15 AM

Object 对象扩展：

1. 对象属性与方法的简易表示法：
   a. 属性名与取值变量名(myvarname)相同时可省略 :myvarname 部分

      ```
      var baz = {foo};  <= 等同于 =>  var baz = {foo: foo};
      ```

   b. 定义方法时可直接省略 :function 部分

      ```
      var obj = { method() { return 'Hello!'; } }  <= 等同于 =>  var obj = { method: function() { return 'Hello!'; } }
      ```

   c. 常见应用：多属性函数返回值

      ```
      function getPoint() { var x = 1, y = 2; return {x, y}; }
      ```

2. 对象属性的字面量定义法(动态生成属性名)：{[myStringExpression]: myValue}

   ```
   var obj = { ['abc'.toUpperCase()]: 'my value' }  <= 等同于 =>  var obj = { ABC: 'my value' }
   ```

3. 对象方法/函数的 name 属性

   ```
   var person = {
    sayName() { console.log(this.name); },
    get firstName() { return 'Nicholas'; },
    lastName: new Function('return "Ryes";')
   };
   var myBoundFunc =  person.sayName.bind();
   var myES6Func =  function() { console.log('此 function 的 name 属性值不同于 ES5'); };
   console.log(person.sayName.name); // 'sayName'
   console.log(person.firstName.name); // 'get firstName' ！！Babel未能验证！！
   console.log(person.lastName.name); // 'anonymous'
   console.log(myBoundFunc.name); // 'bound sayName'
   console.log(myES6Func.name); // 'myES6Func'
   ```

4. 使用 Object.is 判定值是否相等（基本同于 === 运算符）

   ```
   Object.is('foo', 'foo') // true, 判定结果等同于 === 运算符
   Object.is({}, {}) // false, 判定结果等同于 === 运算符
   +0 === -0 // true
   Object.is(+0, -0) // false，特殊例：判定结果不等于 === 运算符
   NaN === NaN // false
   Object.is(NaN, NaN) // true，特殊例：判定结果不等于 === 运算符
   ```

5. 使用 Object.assign 复制对象属性: myTarget = Object.assign(target, ...sources)

   ```
   var myObj1 = {a: 1, b: 2}, myObj2 = {b: '二', c: '三'};
   Object.assign(myObj1, {b: 'two'}); // myObj1 => {a: 1, b: 'two'}
   var myObj3 = Object.assign(myObj1, myObj2, {c: 'III'}); // myObj1 => {a: 1, b: "二", c: "III"}, myObj2 => {b: "二", c: "三"}, myObj3 => {a: 1, b: "二", c: "III"}
   myObj3 === myObj1; // true, Object.assign返回值(myTarget)就是其第一个参数值引用(target)
   Object.assign(null); // 报错 Uncaught TypeError, target 不是 Object
   Object.assign(undefined, {b: 2}); // 报错 Uncaught TypeError
   Object.assign({}, 'abc', true, 8, null, 'de'); // {0: "d", 1: "e", 2: "c"}, 非Object类型的sources元素只有String类型可被枚举并参与赋值到target, 其余类型被忽略
   var myObj4 = {c: 'C', d: 'D'};
   myObj4.__proto__ = myObj2;
   Object.defineProperty(myObj4, 'myInvisibleProperty', {enumerable: false, value: 'hello'});
   var myObj5 = Object.assign({}, myObj4); // myObj5 => {c: "C", d: "D"}, 不拷贝继承属性和不可枚举(enumerable)属性
   var myObj6 = {a: {b: '2', c: '3', d: '4'}};
   var myObj7 = {a: {b: 'two', c: 'three'}};
   Object.assign(myObj6, myObj7); // myObj6 => {a: {b: "two", c: "three"}}, 只执行浅拷贝
   var myObj8 = Object.assign([1,2,3], [4,5]); // [4, 5, 3]
   ```

6. Object.assign 的应用
   a. 为对象添加属性/方法

      ```
      class Point {
       constructor(x, y) {
         Object.assign(this, {x, y});
       }
      }
      Object.assign(Point.prototype, {
       render() { console.log('rendered'); }
      });
      ```

   b. 克隆对象

      ```
      function clone(origin) {
       return Object.assign({}, origin); // 此处仅为浅拷贝，且没有拷贝继承与不可枚举属性
      }
      ```

c. 合并多个对象

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

d. 为对象属性指定默认值

```
const DEFAULTS = {
 logLevel: 0,
 outputFormat: 'html'
};
function processContent(options) {
 let options = Object.assign({}, DEFAULTS, options);
}
```

7. Object 属性的描述对象（Descriptor）及遍历:
   a. 可枚举性(enumerable)：
      i. 定义属性时默认被设定为 true

```
Object.getOwnPropertyDescriptor({abc: 123, def: 345}, 'abc').enumerable // true
Object.getOwnPropertyDescriptor(Object.prototype, 'toString').enumerable // false
```

      ii. 被设定为 false 时在 for... in, Object.keys(), JSON.stringify(), Object.assign() 遍历中会被忽略

   b. 属性遍历：
      i. 使用 for ... in 遍历（包括 String 属性 + 继承属性，不包括 Symbol 属性 + 不可枚举属性）
      ii. 使用 Object.keys(myObj) 遍历并返回一个数组（包括 String 属性，不包括继承属性 + Symbol 属性 + 不可枚举属性）
      iii. 使用 Object.getOwnPropertyNames(myObj) 遍历并返回一个数组（包括 String 属性 + 不可枚举属性，不包括 Symbol 属性 + 继承属性）
      iv. 使用 Object.getOwnPropertySymbols(myObj) 遍历并返回一个数组（包括 Symbol 属性 + 不可枚举属性，不包括 String 属性 + 继承属性）
      v. 使用 Reflect.ownKeys(myObj) 遍历并返回一个数组（包括 String 属性 + Symbol 属性 + 不可枚举属性，不包括继承属性）
      vi. 使用 JSON.stringify(myObj) 遍历并返回一个字符串（包括 String 属性，不包括 Symbol 属性 + 不可枚举属性 + 继承属性）
      vii. 图示：

	String/Number 类型属性名	Symbol 类型属性名	不可枚举属性 (enumerable == false)	继承属性 (eg a.__proto__ = b)
for ... in 遍历	含	不含	不含	含
Object.keys() 数组	含	不含	不含	不含
Object.getOwnPropertyNames() 数组	含	不含	含	不含
Object.getOwnPropertySymbols() 数组	不含	含	含	不含
Reflect.ownKeys() 数组	含	含	含	不含
JSON.stringify() 字符串	含	不含	不含	不含

      viii. Object 属性的通用遍历顺序：Number 属性名(数字排序) => String 属性名(生成时间排序) => Symbol 属性名(生成时间排序)

```
Reflect.ownKeys({ [Symbol()]:0, b:0, 10:0, 2:0, a:0 }); // ['2', '10', 'b', 'a', Symbol()]
```

8. 对象的 prototype 原型操作：
   a. 不建议使用 __proto__ 属性操作对象原型，该属性不属于 ES6 标准，非浏览器环境或未被支持
   b. 使用 Object.setPrototypeOf() 设定原型，Object.getPrototypeOf() 获取原型

```
var myObjBase = {a: 1, b: 2}, myObjA = {b: '二', c: '三'};

var myObjB = Object.setPrototypeOf(myObjA, myObjBase); // myObjA === myObjB --> {a: 1, b: "二", c: "三"}
console.log(myObjBase === Object.getPrototypeOf(myObjB)); // true
```

Array 数组对象扩展：

1. 使用 Array.from() 将类数组对象(含length属性)和可迭代对象(Iterable)转换为数组：

```
var arrayLike = {'0': 'a', '1': 'b', '2': 'c', length: 3};
var arrayLike2 = {length: 2};
var iterable = new Set(['A', 'B']);
console.log(Array.from(arrayLike), Array.from(arrayLike2), Array.from(iterable)); // ["a", "b", "c"], [undefined, undefined], ["A", "B"]
console.log(Array.from([1, 2, 3], x => x * x)); // [1, 4, 9] -> 第二个参数用来对每一个元素进行处理并赋予函数返回值
console.log(Array.from({length: 4}, ()=>'Hi'); // ["Hi", "Hi", "Hi", "Hi"] -> 应用：统一设定数组元素初始值
console.log(Array.from({length: 4}, (value, index)=>index)); // [0, 1, 2, 3] -> 应用：快速生成顺序数值
```

2. 使用 Array.of() 将传入的参数转换为数组：Array.of(...elements)

```
Array.of(3, 11, 8); // [3, 11, 8]
Array.of(); // []
```

3. 使用 Array.prototype.copyWithin() 在当前数组内部复制元素：Array.prototype.copyWithin(targetIndex, startIndex = 0, endIndex = this.length)

```
[5, 6, 7, 8, 9].copyWithin(0, 2); // [7, 8, 9, 8, 9]
[5, 6, 7, 8, 9].copyWithin(0, 2, 4); // [7, 8, 7, 8, 9]
[5, 6, 7, 8, 9].copyWithin(0, -2, -1); // [8, 6, 7, 8, 9]
[5, 6, 7, 8, 9].copyWithin(-2, -3); // [5, 6, 7, 7, 8]
```

4. 使用 Array.prototype.find()/Array.prototyp.findIndex() 查找数组元素：

```
[1, 4, -5, 10].find((n) => n < 0); // -5 -> 找到第一个匹配元素并返回之
[1, 5, 10, 15].findIndex(function(value, index, arr){ return value > 9; }); // 2 -> 找到第一个匹配元素并返回其索引
```

5. 使用 Array.prototype.fill() 给数组填充值：Array.prototype.fill(value, startIndex, endIndex)

```
[5, 6, 7, 8, 9].fill(3); // [3, 3, 3, 3, 3]
[5, 6, 7, 8, 9].fill(3, 1, 3); // [5, 3, 3, 8, 9]
```

6. 使用 Array.prototype.entries()/Array.prototype.keys()/Array.prototype.values() 遍历数组：

```
for (let index of ['a', 'b'].keys()) { console.log(index); } // 0 // 1
for (let elem of ['a', 'b'].values()) { console.log(elem); } // 'a' // 'b'
for (let [index, elem] of ['a', 'b'].entries()) { console.log(index, elem); } // 0, "a" // 1, "b"
```

7. ES6 中数组的空位被转化为 undefined：

```
Array.from(['a',,'b']) // [ "a", undefined, "b" ]
```

8. 使用 ES7 提案之 Array.prototype.includes() 确定数组是否包含传入的参数值：Array.prototype.includes(value, startIndex, endIndex)

```
[1, 2, 3].includes(2);     // true
[1, 2, 3].includes(4);     // false
[1, 2, NaN].includes(NaN); // true
[1, 2, 3, 4].includes(2, 3);  // false
[1, 2, 3, 4].includes(2, -3); // true
```