ES6 PPT1607011: Reflect 和 Proxy 对象

Thursday, November 5, 2015 10:15 AM

Reflect 对象：

1. Reflect 对象下重新封装了一些 Object 对象下的方法或与 Object 相关的操作，这些方法均可与 Proxy 支持的拦截操作一一对应
   a. 属性读取：Reflect.get(target, propKey, receiver) // 取 target 的 propkey 属性值。如果 propKey 部署了读取函数，则该函数的 this 绑定 receiver

```
var myObjGetMember = {
  get browserLanguage() { return this.language; },
  language: 'en-US',
  hello: '哈喽'
};
var myObjGetReceiver = {
  language: 'zh-CN',
  hello: '你好'
}
console.log(Reflect.get(myObjGetMember, 'browserLanguage')); // "en-US"
console.log(Reflect.get(myObjGetMember, 'browserLanguage', myObjGetReceiver)); // "zh-CN"
console.log(Reflect.get(myObjGetMember, 'hello')); // "哈喽"
console.log(Reflect.get(myObjGetReceiver, 'hello', myObjGetReceiver)); // "你好"
```

   b. 属性赋值：Reflect.set(target, propKey, value, receiver) // 设定 target 的 propkey 属性值。如果 propKey 部署了赋值函数，则该函数的 this 绑定 receiver

```
var myObjSetMember = {
  set programLanguage(value) { this.language = value; return true; },
  hello: '哈喽'
};
var myObjSetReceiver = {};
console.log(Reflect.set(myObjSetMember, 'programLanguage', 'Java')); // true -> 设置成功时返回true
console.log(myObjSetMember.programLanguage); // undefined
console.log(myObjSetMember.language); // "Java"
console.log(myObjSetReceiver.language); // undefined
console.log(Reflect.set(myObjSetMember, 'programLanguage', 'JavaScript', myObjSetReceiver)); // true
console.log(myObjSetMember.programLanguage); // undefined
console.log(myObjSetMember.language); // "Java"
console.log(myObjSetReceiver.language); // "JavaScript"
console.log(Reflect.set(myObjSetMember, 'hello', '你好')); // true
console.log(myObjSetMember.hello); // "你好"
console.log(myObjSetReceiver.hello); // undefined
console.log(Reflect.set(myObjSetMember, 'hello', '你好', myObjSetReceiver)); // true
console.log(myObjSetMember.hello); // "你好"
console.log(myObjSetReceiver.hello); // "你好"
```

   c. 判断属性存在：Reflect.has(target, propKey) // 等同于 propKey in target
   d. 获取所有属性名：Reflect.ownKeys(target) // String, Number, Symbol 类型属性名均包括，且无论是否为可枚举属性
   e. 删除属性：Reflect.deleteProperty(target, propKey) // 等同于 delete target[propkey]
   f. 添加新属性：Reflect.defineProperty(target, propKey, propDesc) // 基本同于 Object 下同名方法，但不会抛错而是返回布尔值反应成功与失败

```
console.log(Object.defineProperty({}, "x", {value: 7})); // {x: 7}
console.log(Reflect.defineProperty({}, "y", {value: 8})); // true
```

   g. 获取属性描述对象：Reflect.getOwnPropertyDescriptor(target, propKey) // 基本同于 Object 下同名方法，但 target 为非 object/function 类型时会报错

```
console.log(Object.getOwnPropertyDescriptor("foo", 0)); // {value: "f", writable: false, enumerable: true, configurable: false}
console.log(Reflect.getOwnPropertyDescriptor("foo", 0)); // Uncaught TypeError: Reflect.getOwnPropertyDescriptor called on non-object
```

   h. 防止添加新属性/扩展操作：Reflect.preventExtensions(target) // 基本同于 Object 下同名方法，但 target 为非 object/function 类型时会报错

```
console.log(Object.preventExtensions('foo')); // true
console.log(Reflect.preventExtensions('foo')); // Uncaught TypeError: Reflect.preventExtensions called on non-object
var myObjExt = {hello: '哈喽'};
var myObjProto = {encode: 'UTF8'};
myObjExt.__proto__ = myObjProto;
console.log(Reflect.preventExtensions(myObjExt)); // true
myObjExt.hello = '你好';
console.log(myObjExt.hello); // "你好"
```

```
myObjExt.bye = '拜拜'; // Uncaught TypeError: Can't add property bye, object is not extensible
myObjExt.__proto__ = {}; // Uncaught TypeError: #<Object> is not extensible -> 继承原型proto指向也不可再修改
```

i. 获取能否添加新属性/扩展操作：Reflect.isExtensible(target) // 基本同于 Object 下同名方法，但 target 为非 object/function 类型时会报错

```
console.log(Object.isExtensible('foo')); // false
console.log(Reflect.isExtensible('foo')); // Uncaught TypeError: Reflect.isExtensible called on non-object
var myObjExt2 = {hello: '哈喽'};
console.log(Reflect.isExtensible(myObjExt2)); // true -> 对象Object默认情况下可以扩展
console.log(Reflect.preventExtensions(myObjExt2)); // true -> 成功设置防止扩展
console.log(Reflect.isExtensible(myObjExt2)); // false -> 该对象不可再扩展
var myObjSealed = Object.seal({});
console.log(Reflect.isExtensible(myObjSealed)); // false -> 被密封seal的对象不可扩展
var myObjFrozen = Object.freeze({});
console.log(Reflect.isExtensible(myObjFrozen)); // false -> 被冻结freeze的对象不可扩展
```

j. 获取原型 __proto__ 操作：Reflect.getPrototypeOf(target) // 等同于 Object 下同名方法

k. 修改原型 __proto__ 操作：Reflect.setPrototypeOf(target, proto) // 等同于 Object 下同名方法

l. apply 运行操作：Reflect.apply(target, object, args) // 等同于 Function.prototype.apply.call(target, object, args)，可应对 target.apply 方法被污染

m. new 命令操作：Reflect.construct(target, args) // 等同于 new target(...args)

Proxy 对象：

1. Proxy 意为代理，其使用可以理解成在目标对象前架设一层拦截，外界对该对象的一些操作访问，均通过这层拦截进行
2. 基本语法： var myProxiedTarget = new Proxy(myTarget, myHandler);
   a. 以上 myHandler 对象中定义的拦截方法将作用于从目标对象 myTarget 代理生成的 myProxiedTarget 对象上

```
var myObj = {time: '7pm'};
var myPrxObj = new Proxy(myObj, {
  get: function(target, property) {
    return '我是一个属性';
  },
  set: function(target, property, value) {
    console.log(`设置属性${property}的值`);
    return true; // 设置成功时返回true
  }
});
console.log(myObj.time) // "7pm"
console.log(myPrxObj.time) // "我是一个属性"
console.log(myPrxObj.title) // "我是一个属性"
console.log(myPrxObj.__proto__) // "我是一个属性"
myPrxObj.time='8pm'; // "设置属性time的值"
```

3. Proxy 的 myHandler 对象支持的拦截方法（均可对应 Reflect 下同名方法）：
   a. 拦截属性读取操作：get(target, propKey, receiver)

```
var myObjPerson = {name: "张三"};
myObjPerson.__proto__ = {type: '人'};
var myPrxObjPerson = new Proxy(myObjPerson, {
  get: function(target, propKey) {
    return (propKey in target) ? `属性${propKey}值为: ${Reflect.get(...arguments)}` : `属性${propKey}不存在`;
  }
});
console.log(myPrxObjPerson.name); // "属性name值为: 张三"
console.log(myPrxObjPerson.age); // "属性age不存在"
console.log(myPrxObjPerson.type); // "属性type值为: 人"
// 以下生成一个可通过负索引从后往前取元素值的数组对象
function mySuperArray(...elements) {
  return new Proxy([...elements], {
    get(target, propKey, receiver) {
      let index = Number(propKey);
      propKey = index < 0 ? String(target.length + index) : propKey;
      return Reflect.get(target, propKey, receiver);
    }
  });
}
let arr = mySuperArray('a', 'b', 'c', 'd', 'e', 'f');
console.log(arr[-1], arr[-2], arr[1]); // "f", "e", "b"
```

   b. 拦截属性赋值操作：set(target, propKey, value, receiver)

```
var myObjCat = {name: 'Tom'};
```

```
myObjCat.__proto__ = {type: '猫'};
let myPrxObjCat = new Proxy(myObjCat, {
  set: function(target, propKey, value) {
    if (propKey === 'age') {
      if (!Number.isInteger(value)) {
        console.log(`属性${propKey}的值必须为整数`);
      }
      else {
        Reflect.set(...arguments);
        console.log(`属性${propKey}的值被设定为${value}`);
      }
    }
    return true;
  }
});
myPrxObjCat.age = 3; // "属性age的值被设定为3"
myPrxObjCat.age = '三岁'; // "属性age的值必须为整数"
myPrxObjCat.name = '哆啦A梦';
console.log(myPrxObjCat.name); // "Tom"
```

c. 拦截判断属性存在操作：has(target, propKey)
d. 拦截获取属性名相关的操作：ownKeys(target)
e. 拦截删除属性操作：deleteProperty(target, propKey)
f. 拦截添加新属性操作：defineProperty(target, propKey, propDesc)
g. 拦截获取属性描述对象操作：getOwnPropertyDescriptor(target, propKey)
h. 拦截 Object.preventExtensions() 操作：preventExtensions(target)
i. 拦截 Object.isExtensible() 操作：isExtensible(target)
j. 拦截获取原型 __proto__ 操作：getPrototypeOf(target)
k. 拦截修改原型 __proto__ 操作：setPrototypeOf(target, proto)
l. 拦截 apply 运行操作（target 为 function 类型）：apply(target, object, args)
m. 拦截 new 命令/作为构造函数的操作（target 为 function 类型）：construct(target, args)

4. 调用 Proxy.revocable() 生成一个对象，该对象包含一个指向由其生成的 Proxy 实例的属性以及一个可取消该实例的 revoke 方法

```
let target = {tempVal: '2秒后就取不到我的值了'};
let handler = {};
let myRevocable = Proxy.revocable(target, handler);
console.log(myRevocable.proxy.tempVal); // "2秒后就取不到我的值了"
setTimeout(()=>{
  myRevocable.revoke(); // 收回该Proxy实例
  console.log(myRevocable.proxy.tempVal); // Uncaught TypeError: Cannot perform 'get' on a proxy that has been revoked -> 已被回收，无法取值
}, 2000);
```