ECMAScript 6 入门

作者：阮一峰

授权：署名-非商用许可证

目录

0. 前言
1. ECMAScript 6简介
2. let和const命令
3. 变量的解构赋值
4. 字符串的扩展
5. 正则的扩展
6. 数值的扩展
7. 数组的扩展
8. 函数的扩展
9. 对象的扩展
10. Symbol
11. Proxy和Reflect
12. 二进制数组
13. Set和Map数据结构
14. Iterator和for...of循环
15. Generator函数
16. Promise对象
17. 异步操作和Async函数
18. Class
19. Decorator
20. Module
21. 编程风格
22. 读懂规格
23. 参考链接

其他

- 源码
- 修订历史
- 反馈意见

Proxy和Reflect

back to top
edit

1. Proxy概述
2. Proxy实例的方法
3. Proxy.revocable()
4. Reflect概述
5. Reflect对象的方法

1. Proxy概述

Proxy用于修改某些操作的默认行为，等同于在语言层面做出修改，所以属于一种"元编程"（meta programming），即对编程语言进行编程。

Proxy可以理解成，在目标对象之前架设一层"拦截"，外界对该对象的访问，都必须先通过这层拦截，因此提供了一种机制，可以对外界的访问进行过滤和改写。Proxy这个词的原意是代理，用在这里表示由它来"代理"某些操作，可以译为"代理器"。

```
var obj = new Proxy({}, {
  get: function (target, key, receiver) {
    console.log(`getting ${key}!`);
    return Reflect.get(target, key, receiver);
  },
  set: function (target, key, value, receiver) {
    console.log(`setting ${key}!`);
    return Reflect.set(target, key, value, receiver);
  }
});
```

上面代码对一个空对象架设了一层拦截，重定义了属性的读取（`get`）和设置（`set`）行为。这里暂时先不解释具体的语法，只看运行结果。对设置了拦截行为的对象`obj`，去读写它的属性，就会得到下面的结果。

```
obj.count = 1
//  setting count!
++obj.count
```

```
//  getting count!
//  setting count!
//  2
```

上面代码说明，Proxy实际上重载（overload）了点运算符，即用自己的定义覆盖了语言的原始定义。

ES6原生提供Proxy构造函数，用来生成Proxy实例。

```
var proxy = new Proxy(target, handler);
```

Proxy对象的所有用法，都是上面这种形式，不同的只是`handler`参数的写法。其中，`new Proxy()`表示生成一个Proxy实例，target参数表示所要拦截的目标对象，`handler`参数也是一个对象，用来定制拦截行为。

下面是另一个拦截读取属性行为的例子。

```
var proxy = new Proxy({}, {
  get: function(target, property) {
    return 35;
  }
});

proxy.time // 35
proxy.name // 35
proxy.title // 35
```

上面代码中，作为构造函数，Proxy接受两个参数。第一个参数是所要代理的目标对象（上例是一个空对象），即如果没有Proxy的介入，操作原来要访问的就是这个对象；第二个参数是一个配置对象，对于每一个被代理的操作，需要提供一个对应的处理函数，该函数将拦截对应的操作。比如，上面代码中，配置对象有一个`get`方法，用来拦截对目标对象属性的访问请求。`get`方法的两个参数分别是目标对象和所要访问的属性。可以看到，由于拦截函数总是返回`35`，所以访问任何属性都得到`35`。

注意，要使得Proxy起作用，必须针对Proxy实例（上例是proxy对象）进行操作，而不是针对目标对象（上例是空对象）进行操作。

如果`handler`没有设置任何拦截，那就等同于直接通向原对象。

```
var target = {};
var handler = {};
var proxy = new Proxy(target, handler);
proxy.a = 'b';
target.a // "b"
```

上面代码中，`handler`是一个空对象，没有任何拦截效果，访问`handeler`就等同于访问`target`。

一个技巧是将Proxy对象，设置到`object.proxy`属性，从而可以在`object`对象上调用。

```
var object = { proxy: new Proxy(target, handler) };
```

Proxy实例也可以作为其他对象的原型对象。

```
var proxy = new Proxy({}, {
  get: function(target, property) {
    return 35;
  }
});

let obj = Object.create(proxy);
obj.time // 35
```

上面代码中，`proxy`对象是`obj`对象的原型，`obj`对象本身并没有`time`属性，所以根据原型链，会在`proxy`对象上读取该属性，导致被拦截。

同一个拦截器函数，可以设置拦截多个操作。

```
var handler = {
  get: function(target, name) {
    if (name === 'prototype') {
      return Object.prototype;
    }
    return 'Hello, ' + name;
  },

  apply: function(target, thisBinding, args) {
    return args[0];
  },
```

```
  construct: function(target, args) {
    return {value: args[1]};
  }
};

var fproxy = new Proxy(function(x, y) {
  return x + y;
}, handler);

fproxy(1, 2) // 1
new fproxy(1,2) // {value: 2}
fproxy.prototype === Object.prototype // true
fproxy.foo // "Hello, foo"
```

下面是Proxy支持的拦截操作一览。

对于可以设置、但没有设置拦截的操作，则直接落在目标对象上，按照原先的方式产生结果。

（1）get(target, propKey, receiver)

拦截对象属性的读取，比如`proxy.foo`和`proxy['foo']`。

最后一个参数`receiver`是一个对象，可选，参见下面`Reflect.get`的部分。

（2）set(target, propKey, value, receiver)

拦截对象属性的设置，比如`proxy.foo = v`或`proxy['foo'] = v`，返回一个布尔值。

（3）has(target, propKey)

拦截`propKey in proxy`的操作，以及对象的`hasOwnProperty`方法，返回一个布尔值。

（4）deleteProperty(target, propKey)

拦截`delete proxy[propKey]`的操作，返回一个布尔值。

（5）ownKeys(target)

拦

截`Object.getOwnPropertyNames(proxy)`、`Object.getOwnPropertySymbols(proxy)`、`Object.keys(proxy)`，返回一个数组。该方法返回对象所有自身的属性，而`Object.keys()`仅返回对象可遍历的属性。

（6）getOwnPropertyDescriptor(target, propKey)

拦截`Object.getOwnPropertyDescriptor(proxy, propKey)`，返回属性的描述对象。

（7）defineProperty(target, propKey, propDesc)

拦截`Object.defineProperty(proxy, propKey, propDesc)`、`Object.defineProperties(proxy, propDescs)`，返回一个布尔值。

（8）preventExtensions(target)

拦截`Object.preventExtensions(proxy)`，返回一个布尔值。

（9）getPrototypeOf(target)

拦截`Object.getPrototypeOf(proxy)`，返回一个对象。

（10）isExtensible(target)

拦截`Object.isExtensible(proxy)`，返回一个布尔值。

（11）setPrototypeOf(target, proto)

拦截`Object.setPrototypeOf(proxy, proto)`，返回一个布尔值。

如果目标对象是函数，那么还有两种额外操作可以拦截。

（12）apply(target, object, args)

拦截Proxy实例作为函数调用的操作，比如`proxy(...args)`、`proxy.call(object, ...args)`、`proxy.apply(...)`。

（13）construct(target, args)

拦截Proxy实例作为构造函数调用的操作，比如`new proxy(...args)`。

2. Proxy实例的方法

下面是上面这些拦截方法的详细介绍。

get()

`get`方法用于拦截某个属性的读取操作。上文已经有一个例子，下面是另一个拦截读取操作的例子。

```
var person = {
  name: "张三"
};

var proxy = new Proxy(person, {
  get: function(target, property) {
    if (property in target) {
      return target[property];
    } else {
      throw new ReferenceError("Property \"" + property + "\" does not
    }
  }
});

proxy.name // "张三"
proxy.age // 抛出一个错误
```

上面代码表示，如果访问目标对象不存在的属性，会抛出一个错误。如果没有这个拦截函数，访问不存在的属性，只会返回`undefined`。

`get`方法可以继承。

```
let proto = new Proxy({}, {
  get(target, propertyKey, receiver) {
    console.log('GET '+propertyKey);
    return target[propertyKey];
  }
});
```

```
let obj = Object.create(proto);
obj.xxx // "GET xxx"
```

上面代码中，拦截操作定义在Prototype对象上面，所以如果读取`obj`对象继承的属性时，拦截会生效。

下面的例子使用`get`拦截，实现数组读取负数的索引。

```
function createArray(...elements) {
  let handler = {
    get(target, propKey, receiver) {
      let index = Number(propKey);
      if (index < 0) {
        propKey = String(target.length + index);
      }
      return Reflect.get(target, propKey, receiver);
    }
  };

  let target = [];
  target.push(...elements);
  return new Proxy(target, handler);
}

let arr = createArray('a', 'b', 'c');
arr[-1] // c
```

上面代码中，数组的位置参数是`-1`，就会输出数组的倒数最后一个成员。

利用Proxy，可以将读取属性的操作（`get`），转变为执行某个函数，从而实现属性的链式操作。

```
var pipe = (function () {
  return function (value) {
    var funcStack = [];
    var oproxy = new Proxy({} , {
      get : function (pipeObject, fnName) {
        if (fnName === 'get') {
          return funcStack.reduce(function (val, fn) {
            return fn(val);
          },value);
        }
        funcStack.push(window[fnName]);
```

```
        return oproxy;
      }
    });

    return oproxy;
  }
}());

var double = n => n * 2;
var pow    = n => n * n;
var reverseInt = n => n.toString().split("").reverse().join("") | 0;

pipe(3).double.pow.reverseInt.get; // 63
```

上面代码设置Proxy以后，达到了将函数名链式使用的效果。

下面的例子则是利用`get`拦截，实现一个生成各种DOM节点的通用函数`dom`。

```
const dom = new Proxy({}, {
  get(target, property) {
    return function(attrs = {}, ...children) {
      const el = document.createElement(property);
      for (let prop of Object.keys(attrs)) {
        el.setAttribute(prop, attrs[prop]);
      }
      for (let child of children) {
        if (typeof child === 'string') {
          child = document.createTextNode(child);
        }
        el.appendChild(child);
      }
      return el;
    }
  }
});

const el = dom.div({},
  'Hello, my name is ',
  dom.a({href: '//example.com'}, 'Mark'),
  '. I like:',
  dom.ul({},
    dom.li({}, 'The web'),
    dom.li({}, 'Food'),
    dom.li({}, '…actually that\'s it')
  )
```

```
);

document.body.appendChild(el);
```

---

set()

`set`方法用来拦截某个属性的赋值操作。

假定`Person`对象有一个`age`属性，该属性应该是一个不大于200的整数，那么可以使用`Proxy`保证`age`的属性值符合要求。

```
let validator = {
  set: function(obj, prop, value) {
    if (prop === 'age') {
      if (!Number.isInteger(value)) {
        throw new TypeError('The age is not an integer');
      }
      if (value > 200) {
        throw new RangeError('The age seems invalid');
      }
    }

    // 对于age以外的属性，直接保存
    obj[prop] = value;
  }
};

let person = new Proxy({}, validator);

person.age = 100;

person.age // 100
person.age = 'young' // 报错
person.age = 300 // 报错
```

上面代码中，由于设置了存值函数`set`，任何不符合要求的`age`属性赋值，都会抛出一个错误。利用`set`方法，还可以数据绑定，即每当对象发生变化时，会自动更新DOM。

有时，我们会在对象上面设置内部属性，属性名的第一个字符使用下划线开头，表示这些属性不应该被外部使用。结合`get`和`set`方法，就可以做到防止这些内部属性被外部

读写。

```
var handler = {
  get (target, key) {
    invariant(key, 'get');
    return target[key];
  },
  set (target, key, value) {
    invariant(key, 'set');
    return true;
  }
};
function invariant (key, action) {
  if (key[0] === '_') {
    throw new Error(`Invalid attempt to ${action} private "${key}" prop
  }
}
var target = {};
var proxy = new Proxy(target, handler);
proxy._prop
// Error: Invalid attempt to get private "_prop" property
proxy._prop = 'c'
// Error: Invalid attempt to set private "_prop" property
```

上面代码中，只要读写的属性名的第一个字符是下划线，一律抛错，从而达到禁止读写内部属性的目的。

---

apply()

`apply`方法拦截函数的调用、call和apply操作。

```
var handler = {
  apply (target, ctx, args) {
    return Reflect.apply(...arguments);
  }
};
```

`apply`方法可以接受三个参数，分别是目标对象、目标对象的上下文对象（`this`）和目标对象的参数数组。

下面是一个例子。

```
var target = function () { return 'I am the target'; };
var handler = {
  apply: function () {
    return 'I am the proxy';
  }
};

var p = new Proxy(target, handler);

p()
// "I am the proxy"
```

上面代码中，变量`p`是Proxy的实例，当它作为函数调用时（`p()`），就会被`apply`方法拦截，返回一个字符串。

下面是另外一个例子。

```
var twice = {
  apply (target, ctx, args) {
    return Reflect.apply(...arguments) * 2;
  }
};
function sum (left, right) {
  return left + right;
};
var proxy = new Proxy(sum, twice);
proxy(1, 2) // 6
proxy.call(null, 5, 6) // 22
proxy.apply(null, [7, 8]) // 30
```

上面代码中，每当执行`proxy`函数（直接调用或`call`和`apply`调用），就会被`apply`方法拦截。

另外，直接调用`Reflect.apply`方法，也会被拦截。

```
Reflect.apply(proxy, null, [9, 10]) // 38
```

has()

`has`方法用来拦截`HasProperty`操作，即判断对象是否具有某个属性时，这个方法会生效。典型的操作就是`in`运算符。

下面的例子使用`has`方法隐藏某些属性，不被`in`运算符发现。

```
var handler = {
  has (target, key) {
    if (key[0] === '_') {
      return false;
    }
    return key in target;
  }
};
var target = { _prop: 'foo', prop: 'foo' };
var proxy = new Proxy(target, handler);
'_prop' in proxy // false
```

上面代码中，如果原对象的属性名的第一个字符是下划线，`proxy.has`就会返回`false`，从而不会被`in`运算符发现。

如果原对象不可配置或者禁止扩展，这时`has`拦截会报错。

```
var obj = { a: 10 };
Object.preventExtensions(obj);
var p = new Proxy(obj, {
  has: function(target, prop) {
    return false;
  }
});

'a' in p // TypeError is thrown
```

上面代码中，`obj`对象禁止扩展，结果使用`has`拦截就会报错。

值得注意的是，`has`方法拦截的是`HasProperty`操作，而不是`HasOwnProperty`操作，即`has`方法不判断一个属性是对象自身的属性，还是继承的属性。由于`for...in`操作内部也会用到`HasProperty`操作，所以`has`方法在`for...in`循环时也会生效。

```
let stu1 = {name: 'Owen', score: 59};
let stu2 = {name: 'Mark', score: 99};

let handler = {
  has(target, prop) {
    if (prop === 'score' && target[prop] < 60) {
      console.log(`${target.name} 不及格`);
      return false;
    }
    return prop in target;
  }
}

let oproxy1 = new Proxy(stu1, handler);
let oproxy2 = new Proxy(stu2, handler);

for (let a in oproxy1) {
  console.log(oproxy1[a]);
}
// Owen
// Owen 不及格

for (let b in oproxy2) {
  console.log(oproxy2[b]);
}
// Mark
// Mark 99
```

上面代码中，`for...in`循环时，`has`拦截会生效，导致不符合要求的属性被排除在`for...in`循环之外。

---

construct()

`construct`方法用于拦截`new`命令，下面是拦截对象的写法。

```
var handler = {
  construct (target, args, newTarget) {
    return new target(...args);
  }
};
```

`construct`方法可以接受两个参数。

- `target`：目标对象
- `args`：构建函数的参数对象

下面是一个例子。

```
var p = new Proxy(function() {}, {
  construct: function(target, args) {
    console.log('called: ' + args.join(', '));
    return { value: args[0] * 10 };
  }
});

new p(1).value
// "called: 1"
// 10
```

`construct`方法返回的必须是一个对象，否则会报错。

```
var p = new Proxy(function() {}, {
  construct: function(target, argumentsList) {
    return 1;
  }
});

new p() // 报错
```

---

deleteProperty()

`deleteProperty`方法用于拦截`delete`操作，如果这个方法抛出错误或者返回`false`，当前属性就无法被`delete`命令删除。

```
var handler = {
  deleteProperty (target, key) {
    invariant(key, 'delete');
    return true;
  }
```

```
};
function invariant (key, action) {
  if (key[0] === '_') {
    throw new Error(`Invalid attempt to ${action} private "${key}" prop
  }
}

var target = { _prop: 'foo' };
var proxy = new Proxy(target, handler);
delete proxy._prop
// Error: Invalid attempt to delete private "_prop" property
```

上面代码中，`deleteProperty`方法拦截了`delete`操作符，删除第一个字符为下划线的属性会报错。

---

defineProperty()

`defineProperty`方法拦截了`Object.defineProperty`操作。

```
var handler = {
  defineProperty (target, key, descriptor) {
    return false;
  }
};
var target = {};
var proxy = new Proxy(target, handler);
proxy.foo = 'bar'
// TypeError: proxy defineProperty handler returned false for property
```

上面代码中，`defineProperty`方法返回`false`，导致添加新属性会抛出错误。

---

getOwnPropertyDescriptor()

`getOwnPropertyDescriptor`方法拦截`Object.getOwnPropertyDescriptor`，返回一个属性描述对象或者`undefined`。

```
var handler = {
  getOwnPropertyDescriptor (target, key) {
    if (key[0] === '_') {
      return;
    }
    return Object.getOwnPropertyDescriptor(target, key);
  }
};
var target = { _foo: 'bar', baz: 'tar' };
var proxy = new Proxy(target, handler);
Object.getOwnPropertyDescriptor(proxy, 'wat')
// undefined
Object.getOwnPropertyDescriptor(proxy, '_foo')
// undefined
Object.getOwnPropertyDescriptor(proxy, 'baz')
// { value: 'tar', writable: true, enumerable: true, configurable: true
```

上面代码中，`handler.getOwnPropertyDescriptor`方法对于第一个字符为下划线的属性名会返回`undefined`。

---

getPrototypeOf()

`getPrototypeOf`方法主要用来拦截`Object.getPrototypeOf()`运算符，以及其他一些操作。

- `Object.prototype.__proto__`
- `Object.prototype.isPrototypeOf()`
- `Object.getPrototypeOf()`
- `Reflect.getPrototypeOf()`
- `instanceof`运算符

下面是一个例子。

```
var proto = {};
var p = new Proxy({}, {
  getPrototypeOf(target) {
    return proto;
  }
```

```
});
Object.getPrototypeOf(p) === proto // true
```

上面代码中，`getPrototypeOf`方法拦截`Object.getPrototypeOf()`，返回`proto`对象。

---

isExtensible()

`isExtensible`方法拦截`Object.isExtensible`操作。

```
var p = new Proxy({}, {
  isExtensible: function(target) {
    console.log("called");
    return true;
  }
});

Object.isExtensible(p)
// "called"
// true
```

上面代码设置了`isExtensible`方法，在调用`Object.isExtensible`时会输出`called`。

这个方法有一个强限制，如果不能满足下面的条件，就会抛出错误。

```
Object.isExtensible(proxy) === Object.isExtensible(target)
```

下面是一个例子。

```
var p = new Proxy({}, {
  isExtensible: function(target) {
    return false;
  }
});

Object.isExtensible(p) // 报错
```

ownKeys()

`ownKeys`方法用来拦截`Object.keys()`操作。

```
let target = {};

let handler = {
  ownKeys(target) {
    return ['hello', 'world'];
  }
};

let proxy = new Proxy(target, handler);

Object.keys(proxy)
// [ 'hello', 'world' ]
```

上面代码拦截了对于`target`对象的`Object.keys()`操作，返回预先设定的数组。

下面的例子是拦截第一个字符为下划线的属性名。

```
let target = {
  _bar: 'foo',
  _prop: 'bar',
  prop: 'baz'
};

let handler = {
  ownKeys (target) {
    return Reflect.ownKeys(target).filter(key => key[0] !== '_');
  }
};

let proxy = new Proxy(target, handler);
for (let key of Object.keys(proxy)) {
  console.log(target[key]);
}
// "baz"
```

preventExtensions()

`preventExtensions`方法拦截`Object.preventExtensions()`。该方法必须返回一个布尔值。

这个方法有一个限制，只有当`Object.isExtensible(proxy)`为`false`（即不可扩展）时，`proxy.preventExtensions`才能返回`true`，否则会报错。

```
var p = new Proxy({}, {
  preventExtensions: function(target) {
    return true;
  }
});

Object.preventExtensions(p) // 报错
```

上面代码中，`proxy.preventExtensions`方法返回`true`，但这时`Object.isExtensible(proxy)`会返回`true`，因此报错。

为了防止出现这个问题，通常要在`proxy.preventExtensions`方法里面，调用一次`Object.preventExtensions`。

```
var p = new Proxy({}, {
  preventExtensions: function(target) {
    console.log("called");
    Object.preventExtensions(target);
    return true;
  }
});

Object.preventExtensions(p)
// "called"
// true
```

---

setPrototypeOf()

`setPrototypeOf`方法主要用来拦截`Object.setPrototypeOf`方法。

下面是一个例子。

```
var handler = {
  setPrototypeOf (target, proto) {
    throw new Error('Changing the prototype is forbidden');
  }
};
var proto = {};
var target = function () {};
var proxy = new Proxy(target, handler);
proxy.setPrototypeOf(proxy, proto);
// Error: Changing the prototype is forbidden
```

上面代码中，只要修改`target`的原型对象，就会报错。

---

3. Proxy.revocable()

Proxy.revocable方法返回一个可取消的Proxy实例。

```
let target = {};
let handler = {};

let {proxy, revoke} = Proxy.revocable(target, handler);

proxy.foo = 123;
proxy.foo // 123

revoke();
proxy.foo // TypeError: Revoked
```

`Proxy.revocable`方法返回一个对象，该对象的`proxy`属性是`Proxy`实例，`revoke`属性是一个函数，可以取消`Proxy`实例。上面代码中，当执行`revoke`函数之后，再访问`Proxy`实例，就会抛出一个错误。

---

4. Reflect概述

`Reflect`对象与`Proxy`对象一样，也是ES6为了操作对象而提供的新API。`Reflect`对象的设计目的有这样几个。

（1） 将`Object`对象的一些明显属于语言内部的方法（比如`Object.defineProperty`），放到`Reflect`对象上。现阶段，某些方法同时在`Object`和`Reflect`对象上部署，未来的新方法将只部署在`Reflect`对象上。

（2） 修改某些Object方法的返回结果，让其变得更合理。比如，`Object.defineProperty(obj, name, desc)`在无法定义属性时，会抛出一个错误，而`Reflect.defineProperty(obj, name, desc)`则会返回`false`。

```
// 老写法
try {
  Object.defineProperty(target, property, attributes);
  // success
} catch (e) {
  // failure
}

// 新写法
if (Reflect.defineProperty(target, property, attributes)) {
  // success
} else {
  // failure
}
```

（3） 让`Object`操作都变成函数行为。某些`Object`操作是命令式，比如`name in obj`和`delete obj[name]`，而`Reflect.has(obj, name)`和`Reflect.deleteProperty(obj, name)`让它们变成了函数行为。

```
// 老写法
'assign' in Object // true

// 新写法
Reflect.has(Object, 'assign') // true
```

（4）`Reflect`对象的方法与`Proxy`对象的方法一一对应，只要是`Proxy`对象的方法，就能在`Reflect`对象上找到对应的方法。这就让`Proxy`对象可以方便地调用对应的`Reflect`方法，完成默认行为，作为修改行为的基础。也就是说，不管`Proxy`怎么修

改默认行为，你总可以在`Reflect`上获取默认行为。

```
Proxy(target, {
  set: function(target, name, value, receiver) {
    var success = Reflect.set(target,name, value, receiver);
    if (success) {
      log('property ' + name + ' on ' + target + ' set to ' + value);
    }
    return success;
  }
});
```

上面代码中，`Proxy`方法拦截`target`对象的属性赋值行为。它采用`Reflect.set`方法将值赋值给对象的属性，然后再部署额外的功能。

下面是另一个例子。

```
var loggedObj = new Proxy(obj, {
  get(target, name) {
    console.log('get', target, name);
    return Reflect.get(target, name);
  },
  deleteProperty(target, name) {
    console.log('delete' + name);
    return Reflect.deleteProperty(target, name);
  },
  has(target, name) {
    console.log('has' + name);
    return Reflect.has(target, name);
  }
});
```

上面代码中，每一个`Proxy`对象的拦截操作（`get`、`delete`、`has`），内部都调用对应的Reflect方法，保证原生行为能够正常执行。添加的工作，就是将每一个操作输出一行日志。

有了`Reflect`对象以后，很多操作会更易读。

```
// 老写法
Function.prototype.apply.call(Math.floor, undefined, [1.75]) // 1

// 新写法
```

```
Reflect.apply(Math.floor, undefined, [1.75]) // 1
```

---

5. Reflect对象的方法

`Reflect`对象的方法清单如下，共13个。

- Reflect.apply(target,thisArg,args)
- Reflect.construct(target,args)
- Reflect.get(target,name,receiver)
- Reflect.set(target,name,value,receiver)
- Reflect.defineProperty(target,name,desc)
- Reflect.deleteProperty(target,name)
- Reflect.has(target,name)
- Reflect.ownKeys(target)
- Reflect.isExtensible(target)
- Reflect.preventExtensions(target)
- Reflect.getOwnPropertyDescriptor(target, name)
- Reflect.getPrototypeOf(target)
- Reflect.setPrototypeOf(target, prototype)

上面这些方法的作用，大部分与`Object`对象的同名方法的作用都是相同的，而且它与`Proxy`对象的方法是一一对应的。下面是对其中几个方法的解释。

（1）Reflect.get(target, name, receiver)

查找并返回`target`对象的`name`属性，如果没有该属性，则返回`undefined`。

如果`name`属性部署了读取函数，则读取函数的this绑定`receiver`。

```
var obj = {
  get foo() { return this.bar(); },
  bar: function() { ... }
};

// 下面语句会让 this.bar()
```

```
// 变成调用 wrapper.bar()
Reflect.get(obj, "foo", wrapper);
```

（2）Reflect.set(target, name, value, receiver)

设置`target`对象的`name`属性等于`value`。如果`name`属性设置了赋值函数，则赋值函数的`this`绑定`receiver`。

（3）Reflect.has(obj, name)

等同于`name in obj`。

（4）Reflect.deleteProperty(obj, name)

等同于`delete obj[name]`。

（5）Reflect.construct(target, args)

等同于`new target(...args)`，这提供了一种不使用`new`，来调用构造函数的方法。

（6）Reflect.getPrototypeOf(obj)

读取对象的`__proto__`属性，对应`Object.getPrototypeOf(obj)`。

（7）Reflect.setPrototypeOf(obj, newProto)

设置对象的`__proto__`属性，对应`Object.setPrototypeOf(obj, newProto)`。

（8）Reflect.apply(fun,thisArg,args)

等同于`Function.prototype.apply.call(fun,thisArg,args)`。一般来说，如果要绑定一个函数的this对象，可以这样写`fn.apply(obj, args)`，但是如果函数定义了自己的`apply`方法，就只能写成`Function.prototype.apply.call(fn, obj, args)`，采用Reflect对象可以简化这种操作。

另外，需要注意的是，`Reflect.set()`、`Reflect.defineProperty()`、`Reflect.freeze()`、`Reflect.seal()`和`Reflect.preventExtensions()`返回一个布尔值，表示操作是否成功。它们对应的Object方法，失败时都会抛出错误。

```
// 失败时抛出错误
Object.defineProperty(obj, name, desc);
// 失败时返回false
Reflect.defineProperty(obj, name, desc);
```

上面代码中，`Reflect.defineProperty`方法的作用与`Object.defineProperty`是一样的，都是为对象定义一个属性。但是，`Reflect.defineProperty`方法失败时，不会抛出错误，只会返回`false`。

留言

上一章 下一章