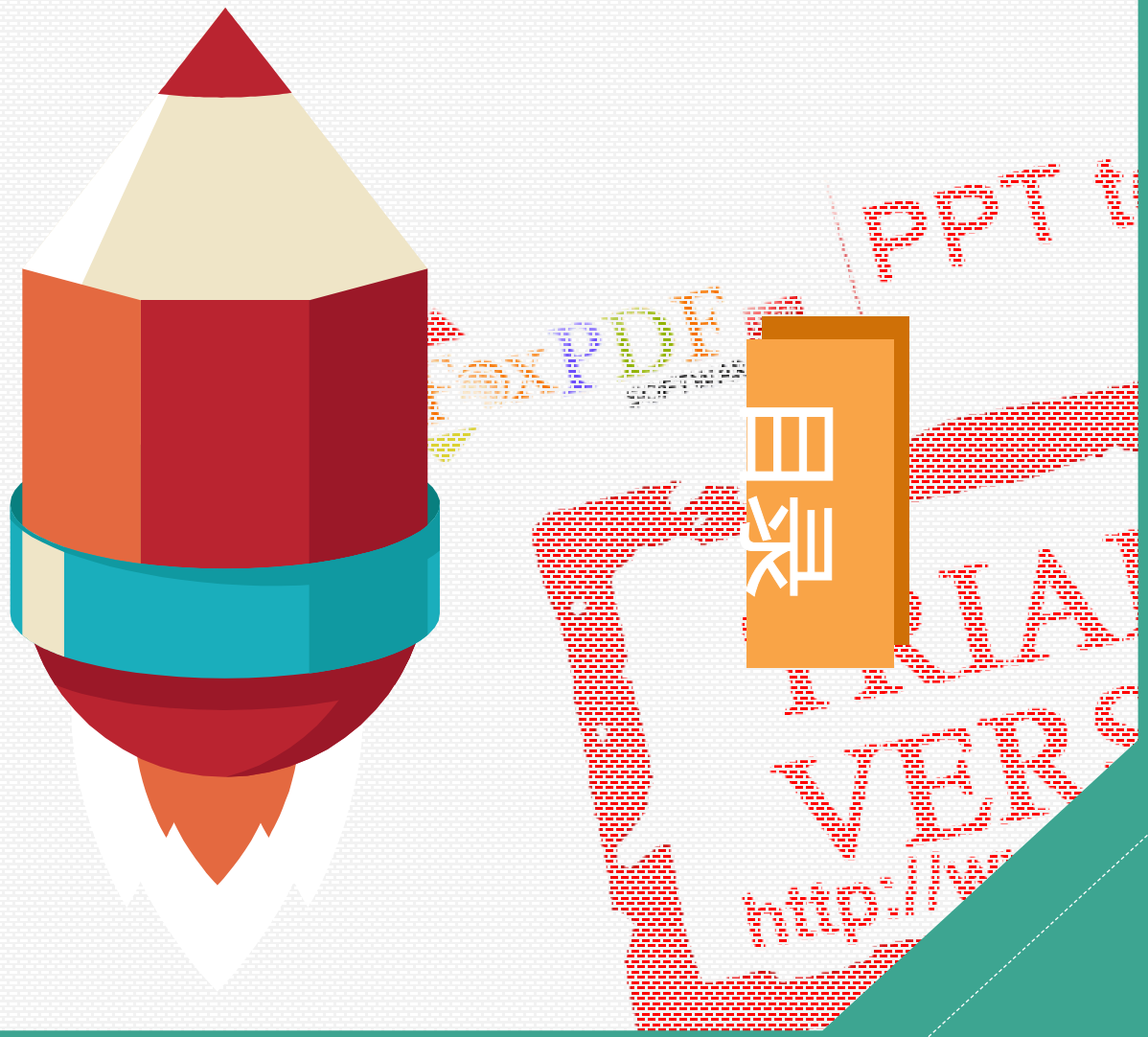

第41课：Proxy
主讲老师：万章



- Proxy的基本概念
- Proxy的拦截操作列表
- Proxy的this问题

Proxy的基本概念

Proxy的基本概念

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 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。

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

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

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

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。

Proxy的基本概念

```
let proxy = new Proxy({
    name: "万章"
}, {
    get: function (target, property,prox) {
        return target[property] + "大帅比";
    },
    set: function (target, property,value,prox) {
        if (property == "name") {
            console.log("你确定要改朕的名字吗?");
            target[property]=value;
        }
    }
})
```

```
> proxy.name
"万章大帅比"
> proxy.name=10;
你确定要改朕的名字吗?                    demo.html:63
10
> proxy.name
"10大帅比"
>
```

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**。

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"
```

Proxy的基本概念

```
> var proxy = new Proxy({}, {
    get: function(target, property) {
      return 35;
    }
  });
< undefined
> proxy
< ▼Proxy {}
    ▼[[Handler]]: Object
      ▶get: ƒ (target, property)
      ▶__proto__: Object
    ▼[[Target]]: Object
      ▶__proto__: Object
     [[IsRevoked]]: false
> let obj = Object.create(proxy);
< undefined
> obj
< ▼{}
    ▼__proto__: Proxy
      ▶[[Handler]]: Object
      ▶[[Target]]: Object
       [[IsRevoked]]: false
>
```

Proxy 实例也可以作为其他对象的原型对象。

Proxy的拦截操作列表

get操作

get(target, propKey, receiver)：拦截对象属性的读取，比如proxy.foo和proxy['foo']。

```
var person = {
    name: "张三"
};

var proxy = new Proxy(person, {
    get: function (target, property) {
        if (property in target) {
            //如果有一个属性，那么就直接返回
            return target[property];
        } else {
            //没有就报错，可以避免传统访问对象未定义的属性名返回undefined的问题
            throw new ReferenceError("Property \"" + property + "\" does not exist.");
        }
    }
});

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

get方法用于拦截某个属性的读取操作，可以接受三个参数，依次为目标对象、属性名和 proxy 实例本身（严格地说，是操作行为所针对的对象，也就是所谓的接收器），其中最后一个参数可选。

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.foo // "GET foo"
```

get操作经典示例之模拟链式操作

```
function pipe(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
```

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。

get的第三个参数

get方法的第三个参数，它总是指向原始的**读操作所在的那个对象（ 简单来说就是到底哪个对象读取了这个属性 ）**，一般情况下就是 Proxy 实例。

```
const proxy = new Proxy({}, {
  get: function(target, property, receiver) {
    return receiver;
  }
});
proxy.getReceiver === proxy // true
```

上面代码中，proxy对象的getReceiver属性是由proxy对象提供的，所以receiver指向proxy对象。

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

const d = Object.create(proxy);
d.a === d // true
```

上面代码中，d对象本身没有a属性，所以读取d.a的时候，会去d的原型proxy对象找。这时，receiver就指向d，代表原始的读操作所在的那个对象。

set操作

```
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');
      }
    }

    // 对于满足条件的 age 属性以及其他属性，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
```

set方法用来拦截某个属性的赋值操作，可以接受四个参数，依次为

1. 目标对象
2. 属性名
3. 属性值
4. Proxy 实例本身

其中最后一个参数可选。

set操作一般用于对于要赋值的数进行过滤,加工或是权限设置

例如右：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。

set例子

```
const handler = {
    get(target, key) {
        invariant(key, 'get');
        return target[key];
    },
    set(target, key, value) {
        invariant(key, 'set');
        target[key] = value;
        return true;
    }
};

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

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。

set注意点

注意，严格模式下，set代理如果没有返回true，就会报错。

```
'use strict';
const handler = {
  set: function(obj, prop, value, receiver) {
    obj[prop] = receiver;
    // 无论有没有下面这一行，都会报错
    return false;
  }
};
const proxy = new Proxy({}, handler);
proxy.foo = 'bar';
// TypeError: 'set' on proxy: trap returned falsish for property 'foo'
```

左侧代码中，严格模式下，set代理返回false或者undefined，都会报错。

apply方法

apply(target, object, args)：三个参数，分别是

1. 目标对象
2. 目标对象的上下文对象（this）
3. 目标对象的参数数组。

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

```
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"
```

has方法

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。（ 但是对for in的操作不起作用 ）

has(target, propKey)：拦截propKey in proxy的操作，返回一个布尔值。
has方法可以接受两个参数，分别是目标对象、需查询的属性名。

```
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
```

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。

construct方法

construct方法用于拦截new命令

construct方法可以接受两个参数。

1. target：目标对象
2. args：构造函数的参数对象

construct方法返回的必须是一个对象，否则会报错。

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

```
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
```

deleteProperty方法

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');
        delete target[key];
        return true;
    }
};

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

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。

注意，目标对象自身的不可配置（configurable）的属性，不能被deleteProperty方法删除，否则报错。

defineProperty方法

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' // 不会生效
```

上面代码中，defineProperty方法返回false，导致添加新属性总是无效。

注意，如果目标对象不可扩展（non-extensible），则defineProperty不能增加目标对象上不存在的属性，否则会报错。另外，如果目标对象的某个属性不可写（writable）或不可配置（configurable），则defineProperty方法不得改变这两个设置。

getOwnPropertyDescriptor方法

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 }
```

getPrototypeOf方法

getPrototypeOf方法主要用来拦截获取对象原型。具体来说，拦截下面这些操作。

- Object.prototype.__proto__
- Object.prototype.isPrototypeOf()
- Object.getPrototypeOf()
- Reflect.getPrototypeOf()
- instanceof

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

左侧代码中，getPrototypeOf方法拦截Object.getPrototypeOf()，返回proto对象。

注意，getPrototypeOf方法的返回值必须是对象或者null，否则报错。另外，如果目标对象不可扩展（non-extensible），getPrototypeOf方法必须返回目标对象的原型对象。

isExtensible()方法

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
```

左侧代码设置了isExtensible方法，在调用Object.isExtensible时会输出called。

注意，该方法只能返回布尔值，否则返回值会被自动转为布尔值。

这个方法有一个强限制，它的返回值必须与目标对象的isExtensible属性保持一致，否则就会抛出错误(如下所示)

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

Object.isExtensible(p)
// Uncaught TypeError: 'isExtensible' on proxy: trap result does not reflect extensibility of proxy target (which is 'true')
```

ownKeys()方法

ownKeys方法用来拦截对象自身属性的读取操作。具体来说，拦截以下操作。

- Object.getOwnPropertyNames()
- Object.getOwnPropertySymbols()
- Object.keys()
- for...in循环

注意，使用Object.keys方法时，有三类属性会被ownKeys方法自动过滤，不会返回。

1. 目标对象上不存在的属性
2. 属性名为 Symbol 值
3. 不可遍历（enumerable）的属性

右侧代码中，ownKeys方法之中，显式返回不存在的属性（d）、Symbol值（Symbol.for('secret')）、不可遍历的属性（key），结果都被自动过滤掉。

```
let target = {
  a: 1,
  b: 2,
  c: 3,
  [Symbol.for('secret')]: '4',
};

Object.defineProperty(target, 'key', {
  enumerable: false,
  configurable: true,
  writable: true,
  value: 'static'
});

let handler = {
  ownKeys(target) {
    return ['a', 'd', Symbol.for('secret'), 'key'];
  }
};

let proxy = new Proxy(target, handler);

Object.keys(proxy)
// ['a']
```

ownKeys()方法

```
var obj = {};

var p = new Proxy(obj, {
  ownKeys: function(target) {
    return [123, true, undefined, null, {}, []];
  }
});

Object.getOwnPropertyNames(p)
// Uncaught TypeError: 123 is not a valid property name
```

ownKeys方法返回的数组成员，只能是字符串或 Symbol 值(虽说Symbol会被忽略)。如果有其他类型的值，或者返回的根本不是数组，就会报错。

如果目标对象自身包含不可配置的属性，则该属性必须被ownKeys方法返回，否则报错。

```
var obj = {};
Object.defineProperty(obj, 'a', {
  configurable: false,
  enumerable: true,
  value: 10 }
);

var p = new Proxy(obj, {
  ownKeys: function(target) {
    return ['b'];
  }
});

Object.getOwnPropertyNames(p)
// Uncaught TypeError: 'ownKeys' on proxy: trap result did not include 'a'
```

deleteProperty方法

```
var obj = {
    a: 1
};

Object.preventExtensions(obj);

var p = new Proxy(obj, {
    ownKeys: function (target) {
        return ['a', 'b'];
    }
});

Object.getOwnPropertyNames(p)
// Uncaught TypeError: 'ownKeys' on proxy: trap returned extra keys but proxy target is non-extensible
```

另外，如果目标对象是不可扩展的（non-extensible），这时ownKeys方法返回的数组之中，必须包含原对象的所有属性，且不能包含多余的属性，否则报错。

preventExtensions方法

preventExtensions方法拦截Object.preventExtensions()。该方法必须返回一个布尔值，否则会被自动转为布尔值。

这个方法有一个限制，只有目标对象不可扩展时（即Object.isExtensible(proxy)为false），proxy.preventExtensions才能返回true，否则会报错。

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

Object.preventExtensions(proxy)
// Uncaught TypeError: 'preventExtensions' on proxy: trap returned truish but the proxy target is extensible
```

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

setPrototypeOf方法

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

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

注意，**该方法只能返回布尔值，否则会被自动转为布尔值**。另外，如果目标对象不可扩展（non-extensible），setPrototypeOf方法不得改变目标对象的原型。

Proxy.revocable()方法

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。

Proxy.revocable的一个使用场景是，目标对象不允许直接访问，必须通过代理访问，一旦访问结束，就收回代理权，不允许再次访问。

Proxy支持的所有拦截操作方法(一共13种)

- get(target, propKey, receiver)：拦截对象属性的读取，比如proxy.foo和proxy['foo']。
- 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。
- has(target, propKey)：拦截propKey in proxy的操作，返回一个布尔值。
- deleteProperty(target, propKey)：拦截delete proxy[propKey]的操作，返回一个布尔值。
- ownKeys(target)：拦截Object.getOwnPropertyNames(proxy)、Object.getOwnPropertySymbols(proxy)、Object.keys(proxy)、for...in循环，返回一个数组。该方法返回目标对象所有自身的属性的属性名，而Object.keys()的返回结果仅包括目标对象自身的可遍历属性。
- getOwnPropertyDescriptor(target, propKey)：拦截Object.getOwnPropertyDescriptor(proxy, propKey)，返回属性的描述对象。

Proxy支持的所有拦截操作方法(一共13种)

- defineProperty(target, propKey, propDesc)：拦截Object.defineProperty(proxy, propKey, propDesc)、Object.defineProperties(proxy, propDescs)，返回一个布尔值。
- preventExtensions(target)：拦截Object.preventExtensions(proxy)，返回一个布尔值。
- getPrototypeOf(target)：拦截Object.getPrototypeOf(proxy)，返回一个对象。
- isExtensible(target)：拦截Object.isExtensible(proxy)，返回一个布尔值。
- 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。
- 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(...)。
- construct(target, args)：拦截 Proxy 实例作为构造函数调用的操作，比如new proxy(...args)。

Proxy的this问题

Proxy的this问题

虽然 Proxy 可以代理针对目标对象的访问，但它不是目标对象的透明代理，**即不做任何拦截的情况下，也无法保证与目标对象的行为一致**。主要原因就是在 Proxy 代理的情况下，目标对象内部的this关键字会指向 Proxy 代理。

```
const target = {
  m: function () {
    console.log(this === proxy);
  }
};
const handler = {};

const proxy = new Proxy(target, handler);

target.m() // false
proxy.m()  // true
```

上面代码中，一旦proxy代理target.m，后者内部的this就是指向proxy，而不是target。

```
const _name = new WeakMap();

class Person {
  constructor(name) {
    _name.set(this, name);
  }
  get name() {
    return _name.get(this);
  }
}

const jane = new Person('Jane');
jane.name // 'Jane'

const proxy = new Proxy(jane, {});
proxy.name // undefined
```

上面代码中，目标对象jane的name属性，实际保存在外部WeakMap对象_name上面，通过this键区分。由于通过proxy.name访问时，this指向proxy，导致无法取到值，所以返回undefined。

Proxy的this问题

```
const target = new Date();
const handler = {};
const proxy = new Proxy(target, handler);

proxy.getDate();
// TypeError: this is not a Date object.
```

有些原生对象的内部属性，只有通过正确的this才能拿到，所以 Proxy 也无法代理这些原生对象的属性。

手动用this绑定原始对象，就可以解决这个问题。

```
const target = new Date('2015-01-01');
const handler = {
  get(target, prop) {
    if (prop === 'getDate') {
      return target.getDate.bind(target);
    }
    return Reflect.get(target, prop);
  }
};
const proxy = new Proxy(target, handler);

proxy.getDate() // 1
```