

Proxy

Proxy 构造函数，用来生成 Proxy 实例。`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。

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

拦截读取属性行为：

```
const proxy = new Proxy({}, {  
  get: (target, propKey) => {  
    return 35;  
  }  
});  
proxy.time // 35  
proxy.name // 35  
proxy.title // 35
```

上例中，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，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。上例中，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。

要使得 Proxy 起作用，必须针对 Proxy 实例进行操作，而不是针对目标对象进行操作。

如果 `handler` 没有设置任何拦截，那就等同于直接通向原对象。

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

上例中，`handler` 是一个空对象，没有任何拦截效果，访问 `proxy` 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。

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

Proxy 实例也可以作为其他对象的原型对象。

```
const proxy = new Proxy({}, {
  get: function(target, propKey) {
    return 35;
  }
});
const obj = Object.create(proxy); // proxy 变成 obj 原型
obj.time // 35
// obj 中没有 time 属性, 从原型 (链) 中查找, 取 proxy 中 time。proxy 中所有属性的值因为
// 被拦截都变成了 35
```

同一个拦截器函数, 可以设置拦截多个操作。

```
const 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]};
  }
};

const proxy = new Proxy(function(x, y) {
  return x + y;
}, handler);

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

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。

2. Proxy 实例的方法

- `get(target, propKey, receiver)` 1 拦截对象属性的读取, 比如 `proxy.foo` 和 `proxy['foo']`。
- `set(target, propKey, value, receiver)` 1 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)` 1 拦截 `propKey in proxy` 的操作, 返回一个布尔值。
- `deleteProperty(target, propKey)` 拦截 `deleteProperty 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。
- `defineProperty(target, propKey, propDesc)` 拦截 `Object.defineProperty(proxy, propKey, propDesc)`、`Object.defineProperties(proxy, propDesc)`，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)`。

2.1. get()

`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），其中最后一个参数可选。

```
const person = { name: "张三" };
const proxy = new Proxy(person, {
  get: function(target, propKey) {
    if (propKey in target) {
      return target[propKey];
    } else {
      throw new ReferenceError("Prop name \"" + propKey + "\" does not exist.");
    }
  }
});
proxy.name; // "张三"
proxy.age;  // ReferenceError: Prop name "age" does not exist.
```

上例表示，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。

```
const proto = new Proxy({}, {
  get(target, propertyKey) {
```

```

    console.log('GET ' + propertyKey);
  }
});
const obj = Object.create(proto); // proto 变成了 obj 的原型
obj.foo; // "GET foo"
// obj 本身没有 foo 属性，会从原型（链）中查找，输出 `GET ${属性名}`

```

使用 `get` 拦截，实现数组读取负数的索引：

```

function createArray(...elements) {
  const handler = {
    get(target, propKey, receiver) {
      const index = Number(propKey); // 将传入的属性变成数值
      if (index < 0) {
        // 如果数值小于 0（负索引），属性被赋值为数组长度加上负索引
        propKey = String(target.length + index);
      }
      return Reflect.get(target, propKey, receiver);
    }
  };
  const target = [];
  target.push(...elements);
  return new Proxy(target, handler);
}
const arr = createArray('a', 'b', 'c');
arr[-1]; // c
// 数组的位置参数是-1，就会输出数组的倒数第一个成员。

```

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。

```

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

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

```

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

上例设置 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 (const prop of Object.keys(attrs)) { el.setAttribute(prop, attrs[prop]); } for (const 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); 下面是一个get方法的第三个参数的例子，它总是指向原始的读操作所在的那个对象，一般情况下就是 Proxy 实例。

```
const proxy = new Proxy({}, { get: function(target, key, receiver) { return receiver; } }); proxy.getReceiver === proxy // true 上例中，proxy对象的getReceiver属性会被get()拦截，得到的返回值就是proxy对象。
```

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

```
const d = Object.create(proxy); d.a === d // true 上例中，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。
```

如果一个属性不可配置（configurable）且不可写（writable），则 Proxy 不能修改该属性，否则通过 Proxy 对象访问该属性会报错。

```
const target = Object.defineProperties({}, { foo: { value: 123, writable: false, configurable: false }, });
```

```
const handler = { get(target, propKey) { return 'abc'; } };
```

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

```
proxy.foo // TypeError: Inconstant check failed
```

2.2. set()

set方法用来拦截某个属性的赋值操作，可以接受四个参数，依次为目标对象、属性名、属性值和 Proxy 实例本身，其中最后一个参数可选。

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。

```
const 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;
```

```
return true;
```

```
});
```

```
const person = new Proxy({}, validator);
```

```
person.age = 100;
```

person.age // 100 person.age = 'young' // 报错 person.age = 300 // 报错 上例中，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，这是数据验证的一种实现方法。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。

```
const handler = { get (target, key) { inconstiant(key, 'get'); return target[key]; }, set (target, key, value) {
inconstiant(key, 'set'); target[key] = value; return true; } }; function inconstiant (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 上例中，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。
```

下面是set方法第四个参数的例子。

```
const handler = { set: function(obj, prop, value, receiver) { obj[prop] = receiver; return true; } }; const proxy =
new Proxy({}, handler); proxy.foo = 'bar'; proxy.foo === proxy // true 上例中，set方法的第四个参数receiver，
指的是原始的操作行为所在的那个对象，一般情况下是proxy实例本身，请看下面的例子。
```

```
const handler = { set: function(obj, prop, value, receiver) { obj[prop] = receiver; return true; } }; const proxy =
new Proxy({}, handler); const myObj = {}; Object.setPrototypeOf(myObj, proxy);
```

myObj.foo = 'bar'; myObj.foo === myObj // true 上例中，设置myObj.foo属性的值时，myObj并没有foo属性，因此引擎会到myObj的原型链去找foo属性。myObj的原型对象proxy是一个 Proxy 实例，设置它的foo属性会触发set方法。这时，第四个参数receiver就指向原始赋值行为所在的对象myObj。

注意，如果目标对象自身的某个属性不可写，那么set方法将不起作用。

```
const obj = {}; Object.defineProperty(obj, 'foo', { value: 'bar', writable: false });
```

```
const handler = { set: function(obj, prop, value, receiver) { obj[prop] = 'baz'; return true; } };

```

```
const proxy = new Proxy(obj, handler); proxy.foo = 'baz'; proxy.foo // "bar" 上例中，obj.foo属性不可写，
Proxy 对这个属性的set代理将不会生效。
```

注意，set代理应当返回一个布尔值。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' 上例中，严格模式下，set代理返回false或者undefined，都会报
错。
```

2.3. 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。

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。

下面的例子使用has()方法隐藏某些属性，不被in运算符发现。

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

```
const obj = { a: 10 }; Object.preventExtensions(obj);
```

```
const p = new Proxy(obj, { has: function(target, prop) { return false; } });
```

'a' in p // TypeError is thrown 上例中，obj对象禁止扩展，结果使用has拦截就会报错。也就是说，如果某个属性不可配置（或者目标对象不可扩展），则has()方法就不得“隐藏”（即返回false）目标对象的该属性。

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。

另外，虽然for...in循环也用到了in运算符，但是has()拦截对for...in循环不生效。

```
const stu1 = {name: '张三', score: 59}; const stu2 = {name: '李四', score: 99};
```

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

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

```
'score' in oproxy1 // 张三 不及格 // false
```

```
'score' in oproxy2 // true
```

```
for (const a in oproxy1) { console.log(oproxy1[a]); } // 张三 // 59
```

for (const b in oproxy2) { console.log(oproxy2[b]); } // 李四 // 99 上例中，has()拦截只对in运算符生效，对for...in循环不生效，导致不符合要求的属性没有被for...in循环所排除。

2.4. 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。

```
const 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); } }
```

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

2.5. ownKeys()

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 下面是拦截Object.keys()的例子。

```
const target = { a: 1, b: 2, c: 3 };
```

```
const handler = { ownKeys(target) { return ['a']; } };
```

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

Object.keys(proxy) // ['a'] 上例拦截了对于target对象的Object.keys()操作，只返回a、b、c三个属性之中的a属性。

下面的例子是拦截第一个字符为下划线的属性名。

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

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

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

"baz" 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。

目标对象上不存在的属性 属性名为 Symbol 值 不可遍历（enumerable）的属性 const target = { a: 1, b: 2, c: 3, [Symbol.for('secret')]: '4', };

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

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

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

Object.keys(proxy) // ['a'] 上例中，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。

ownKeys()方法还可以拦截Object.getOwnPropertyNames()。

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

Object.getOwnPropertyNames(p) // ['a', 'b', 'c'] for...in循环也受到ownKeys()方法的拦截。

```
const obj = { hello: 'world' }; const proxy = new Proxy(obj, { ownKeys: function () { return ['a', 'b']; } });
```

for (const key in proxy) { console.log(key); } // 没有任何输出 上例中，ownkeys()指定只返回a和b属性，由于obj没有这两个属性，因此for...in循环不会有任何输出。

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


```
const obj = {};
```

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

Object.getOwnPropertyNames(p) // Uncaught TypeError: 123 is not a valid property name 上例中，ownKeys()方法虽然返回一个数组，但是每一个数组成员都不是字符串或 Symbol 值，因此就报错了。

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。

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

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

Object.getOwnPropertyNames(p) // Uncaught TypeError: 'ownKeys' on proxy: trap result did not include 'a' 上例中，obj对象的a属性是不可配置的，这时ownKeys()方法返回的数组之中，必须包含a，否则会报错。

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。

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

```
Object.preventExtensions(obj);
```

```
const 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 上例中，obj对象是不可扩展的，这时ownKeys()方法返回的数组之中，包含了obj对象的多余属性b，所以导致了报错。

2.6. getOwnPropertyDescriptor()

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

```
const handler = { getOwnPropertyDescriptor (target, key) { if (key[0] === '_') { return; } return Object.getOwnPropertyDescriptor(target, key); } }; const target = { _foo: 'bar', baz: 'tar' }; const 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 } 上例中，handler.getOwnPropertyDescriptor()方法对于第一个字符为下划线的属性名会返回undefined。
```

2.7. defineProperty()

defineProperty()方法拦截了Object.defineProperty()操作。

```
const handler = { defineProperty (target, key, descriptor) { return false; } }; const target = {}; const proxy = new Proxy(target, handler); proxy.foo = 'bar' // 不会生效 上例中，defineProperty()方法内部没有任何操作，只返回false，导致添加新属性总是无效。注意，这里的false只是用来提示操作失败，本身并不能阻止添加新属性。
```

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。

2.8. preventExtensions()

`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。

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

`Object.preventExtensions(proxy)` // Uncaught TypeError: 'preventExtensions' on proxy: trap returned truthy but the proxy target is extensible 上例中，`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()`。

```
const proxy = new Proxy({}, { preventExtensions: function(target) { console.log('called');
Object.preventExtensions(target); return true; } });
```

```
Object.preventExtensions(proxy) // "called" // Proxy {}
```

2.9. getPrototypeOf()

`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.prototype` `Object.prototype.isPrototypeOf()` `Object.getPrototypeOf()` `Reflect.getPrototypeOf()` `instanceof` 下面是一个例子。

```
const proto = {}; const p = new Proxy({}, { getPrototypeOf(target) { return proto; } }); Object.getPrototypeOf(p)
=== proto // true 上例中，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。

2.10. isExtensible()

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

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

```
Object.isExtensible(p) // "called" // true 上例设置了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。

`Object.isExtensible(proxy) === Object.isExtensible(target)` 下面是一个例子。

```
const 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')
```

2.11. setPrototypeOf()

setPrototypeOf()方法主要用来拦截Object.setPrototypeOf()方法。

下面是一个例子。

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

上例中，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。

2.12. apply()

apply方法拦截函数的调用、call和apply操作。

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。

const handler = { apply (target, ctx, args) { return Reflect.apply(...arguments); } }; 下面是一个例子。

```
const target = function () { return 'I am the target'; }; const handler = { apply: function () { return 'I am the proxy'; } };
const p = new Proxy(target, handler);
```

p() // "I am the proxy" 上例中，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。

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

上例中，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
```

2.13. construct()

construct()方法用于拦截new命令，下面是拦截对象的写法。

const handler = { construct (target, args, newTarget) { return new target(...args); } }; construct()方法可以接受三个参数。

target：目标对象。 args：构造函数的参数数组。 newTarget：创造实例对象时，new命令作用的构造函数（下面例子的p）。 const 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。

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

`new p()` // 报错 // Uncaught TypeError: 'construct' on proxy: trap returned non-object ('1') 另外，由于 `construct()` 拦截的是构造函数，所以它的目标对象必须是函数，否则就会报错。

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

`new p()` // 报错 // Uncaught TypeError: p is not a constructor 上面例子中，拦截的目标对象不是一个函数，而是一个对象（`new Proxy()`的第一个参数），导致报错。

注意，`construct()`方法中的`this`指向的是`handler`，而不是实例对象。

```
const handler = { construct: function(target, args) { console.log(this === handler); return new target(...args); } }
```

```
const p = new Proxy(function () {}, handler); new p() // true
```