```
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。
同一个拦截器函数，可以设置拦截多个操作。
复制代码

```
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 args[1]; }
};

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

fproxy(1,2); // 1
new fproxy(1,2); // 2
fproxy.prototype; // Object.prototype
fproxy.foo; // 'Hello, foo'
```

复制代码

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']，返回类型不限。最后一个参数receiver可选，当target对象设置了propKey属性的get函数时，receiver对象会绑定get函数的this对象。

（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，返回一个布尔值。

has方法可以隐藏某些属性，不被in操作符发现。

（4）deleteProperty(target, propKey)

拦截delete proxy[propKey]的操作，返回一个布尔值。

（5）enumerate(target)

拦截for (var x in proxy)，返回一个遍历器。

注意与Proxy对象的has方法区分，后者用来拦截in操作符，对for...in循环无效。

（6）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。

（7）getOwnPropertyDescriptor(target, propKey)

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

（8）defineProperty(target, propKey, propDesc)

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

（9）preventExtensions(target)

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

（10）getPrototypeOf(target)

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

（11）isExtensible(target)

拦截Object.isExtensible(proxy)，返回一个布尔值。

（12）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。

（13）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(...)。

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。

（14）construct(target, args, proxy)

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

construct方法用于拦截new命令。

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。
复制代码

```
// 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
}
```

复制代码

（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。
Reflect对象的方法

Reflect对象的方法清单如下，共14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.enumerate(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。

（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。