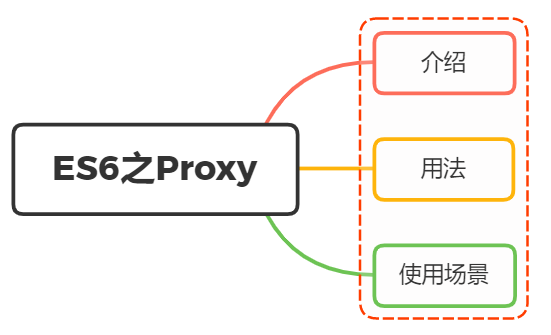
# 面试官：你是怎么理解ES6中Proxy的？使用场景?



## 一、介绍

**定义：** 用于定义基本操作的自定义行为

**本质：** 修改的是程序默认形为，就形同于在编程语言层面上做修改，属于元编程(meta programming)

元编程（Metaprogramming，又译超编程，是指某类计算机程序的编写，这类计算机程序编写或者操纵其它程序（或者自身）作为它们的数据，或者在运行时完成部分本应在编译时完成的工作

一段代码来理解

#!/bin/bash  
# metaprogram  
echo '#!/bin/bash' >program  
for ((I=1; I<=1024; I++)) do  
 echo "echo $I" >>program  
done  
chmod +x program

这段程序每执行一次能帮我们生成一个名为program的文件，文件内容为1024行echo，如果我们手动来写1024行代码，效率显然低效

* 元编程优点：与手工编写全部代码相比，程序员可以获得更高的工作效率，或者给与程序更大的灵活度去处理新的情形而无需重新编译

Proxy 亦是如此，用于创建一个对象的代理，从而实现基本操作的拦截和自定义（如属性查找、赋值、枚举、函数调用等）

## 二、用法

Proxy为 构造函数，用来生成 Proxy实例

var proxy = new Proxy(target, handler)

### 参数

target表示所要拦截的目标对象（任何类型的对象，包括原生数组，函数，甚至另一个代理））

handler通常以函数作为属性的对象，各属性中的函数分别定义了在执行各种操作时代理 p 的行为

### handler解析

关于handler拦截属性，有如下：

* get(target,propKey,receiver)：拦截对象属性的读取
* set(target,propKey,value,receiver)：拦截对象属性的设置
* 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.keys(proxy)、for...in等循环，返回一个数组
* 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），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
* apply(target, object, args)：拦截 Proxy 实例作为函数调用的操作
* construct(target, args)：拦截 Proxy 实例作为构造函数调用的操作

### Reflect

若需要在Proxy内部调用对象的默认行为，建议使用Reflect，其是ES6中操作对象而提供的新 API

基本特点：

* 只要Proxy对象具有的代理方法，Reflect对象全部具有，以静态方法的形式存在
* 修改某些Object方法的返回结果，让其变得更合理（定义不存在属性行为的时候不报错而是返回false）
* 让Object操作都变成函数行为

下面我们介绍proxy几种用法：

### get()

get接受三个参数，依次为目标对象、属性名和 proxy 实例本身，最后一个参数可选

var person = {  
 name: "张三"  
};  
  
var proxy = new Proxy(person, {  
 get: function(target, propKey) {  
 return Reflect.get(target,propKey)  
 }  
});  
  
proxy.name // "张三"

get能够对数组增删改查进行拦截，下面是试下你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

注意：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: Invariant check failed

### 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

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

如果目标对象自身的某个属性，不可写且不可配置，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';  
 }  
};  
  
const proxy = new Proxy(obj, handler);  
proxy.foo = 'baz';  
proxy.foo // "bar"

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'

### 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');  
 Reflect.deleteProperty(target,key)  
 return true;  
 }  
};  
function invariant (key, action) {  
 if (key[0] === '\_') {  
 throw new Error(`无法删除私有属性`);  
 }  
}  
  
var target = { \_prop: 'foo' };  
var proxy = new Proxy(target, handler);  
delete proxy.\_prop  
// Error: 无法删除私有属性

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

### 取消代理

Proxy.revocable(target, handler);

## 三、使用场景

Proxy其功能非常类似于设计模式中的代理模式，常用功能如下：

* 拦截和监视外部对对象的访问
* 降低函数或类的复杂度
* 在复杂操作前对操作进行校验或对所需资源进行管理

使用 Proxy 保障数据类型的准确性

let numericDataStore = { count: 0, amount: 1234, total: 14 };  
numericDataStore = new Proxy(numericDataStore, {  
 set(target, key, value, proxy) {  
 if (typeof value !== 'number') {  
 throw Error("属性只能是number类型");  
 }  
 return Reflect.set(target, key, value, proxy);  
 }  
});  
  
numericDataStore.count = "foo"  
// Error: 属性只能是number类型  
  
numericDataStore.count = 333  
// 赋值成功

声明了一个私有的 apiKey，便于 api 这个对象内部的方法调用，但不希望从外部也能够访问 api.\_apiKey

let api = {  
 \_apiKey: '123abc456def',  
 getUsers: function(){ },  
 getUser: function(userId){ },  
 setUser: function(userId, config){ }  
};  
const RESTRICTED = ['\_apiKey'];  
api = new Proxy(api, {  
 get(target, key, proxy) {  
 if(RESTRICTED.indexOf(key) > -1) {  
 throw Error(`${key} 不可访问.`);  
 } return Reflect.get(target, key, proxy);  
 },  
 set(target, key, value, proxy) {  
 if(RESTRICTED.indexOf(key) > -1) {  
 throw Error(`${key} 不可修改`);  
 } return Reflect.get(target, key, value, proxy);  
 }  
});  
  
console.log(api.\_apiKey)  
api.\_apiKey = '987654321'  
// 上述都抛出错误

还能通过使用Proxy实现观察者模式

观察者模式（Observer mode）指的是函数自动观察数据对象，一旦对象有变化，函数就会自动执行

observable函数返回一个原始对象的 Proxy 代理，拦截赋值操作，触发充当观察者的各个函数

const queuedObservers = new Set();  
  
const observe = fn => queuedObservers.add(fn);  
const observable = obj => new Proxy(obj, {set});  
  
function set(target, key, value, receiver) {  
 const result = Reflect.set(target, key, value, receiver);  
 queuedObservers.forEach(observer => observer());  
 return result;  
}

观察者函数都放进Set集合，当修改obj的值，在会set函数中拦截，自动执行Set所有的观察者

## 参考文献

* https://es6.ruanyifeng.com/#docs/proxy
* https://vue3js.cn/es6