代理与反射

代理捕获器与反射方法

可以捕获13中不同的基本操作

get()

获取属性值操作中被调用
反射API Reflect.get()

```
const myTarget = {};
const proxy = new Proxy(myTarget, {
 get(target, property, receiver) {
 console.log('get()');
 return Reflect.get(...arguments)
 }
});
proxy.foo;
// get()
```

- 返回值无限制
- 拦截的操作
  - proxy.property
  - proxy[property]
  - Object.create(proxy)[property]
  - Reflect.get(proxy,property,receiver)
- 捕获器处理程序对象参数
  - target 目标对象
  - property 引用的目标对象上的字符串键属性
  - receiver 代理对象或继承代理对象的对象
- 捕获器不变式
  如果 target.property 不可写且不可配置，返回值必须与 target.property 匹配
  如果果 target.property 不可配置且[[Get]]特性为 undefined，处理程序的返回值也必须是 undefined

set()

设置属性值的操作被调用
Reflect.set()

```
const myTarget = {};
const proxy = new Proxy(myTarget, {
 set(target, property, value, receiver) {
 console.log('set()');
 return Reflect.set(...arguments)
 }
```

```
});
proxy.foo = 'bar';
// set()
```

- 返回值
  true/false，表示成功或者失败，严格模式失败则报错
- 拦截的操作
  - proxy.property = value
  - proxy[property] = value
  - Object.create(proxy)[property] = value
  - Reflect.set(proxy, property, value, receiver)
- 捕获器处理程序参数
  - target：目标对象。
  - property：引用的目标对象上的字符串键属性
  - value：要赋给属性的值。
  - receiver：接收最初赋值的对象。
- 捕获器不变式
  如果 target.property 不可写且不可配置，则不能修改目标属性的值。
  如果 target.property 不可配置且[[Set]]特性为 undefined，则不能修改目标属性的值。在严格模式下，处理程序中返回 false 会抛出 TypeError

has()

in操作符中被调用
Reflect.has()

```
const myTarget = {};
const proxy = new Proxy(myTarget, {
 has(target, property) {
 console.log('has()');
 return Reflect.has(...arguments)
 }
});
'foo' in proxy;
// has()
```

- 返回值
  布尔值，表示属性是否存在。返回非布尔值会被转型为布尔值。
- 拦截的操作
  - property in proxy
  - property in Object.create(proxy)
  - with(proxy) {(property);}
  - Reflect.has(proxy, property)
- 捕获器处理程序参数
  - target：目标对象。
  - property：引用的目标对象上的字符串键属性。
- 捕获器不变式
  如果 target.property 存在且不可配置，则处理程序必须返回 true
  如果 target.property 存在且目标对象不可扩展，则处理程序必须返回 true

defineProperty()

Object.defineProperty()中被调用
Reflect.defineProperty()

```
const myTarget = {};
const proxy = new Proxy(myTarget, {
 defineProperty(target, property, descriptor) {
 console.log('defineProperty()');
 return Reflect.defineProperty(...arguments)
 }
});
Object.defineProperty(proxy, 'foo', { value: 'bar' });
// defineProperty()
```

- 返回值
  defineProperty()必须返回布尔值，表示属性是否成功定义。返回非布尔值会被转型为布尔值。
- 拦截的操作
  - Object.defineProperty(proxy, property, descriptor)
  - Reflect.defineProperty(proxy, property, descriptor)
- 捕获器处理程序参数
  - target：目标对象。
  - property：引用的目标对象上的字符串键属性
  - descriptor：包含可选的 enumerable、configurable、writable、value、get 和 set 定义的对象。
- 捕获器不变式
  如果目标对象不可扩展，则无法定义属性
  如果目标对象有一个可配置的属性，则不能添加同名的不可配置属性
  如果目标对象有一个不可配置的属性，则不能添加同名的可配置属性

getOwnPropertyDescriptor()

Object.getOwnPropertyDescriptor()中被调用
Reflect.getOwnPropertyDescriptor()

```
const myTarget = {};
const proxy = new Proxy(myTarget, {
 getOwnPropertyDescriptor(target, property) {
 console.log('getOwnPropertyDescriptor()');
 return Reflect.getOwnPropertyDescriptor(...arguments)
 }
});
Object.getOwnPropertyDescriptor(proxy, 'foo');
// getOwnPropertyDescriptor()
```

- 返回值
  getOwnPropertyDescriptor()必须返回对象，或者在属性不存在时返回 undefined
- 拦截的操作
  - Object.getOwnPropertyDescriptor(proxy, property)
  - Reflect.getOwnPropertyDescriptor(proxy, property)
- 捕获器处理程序参数

    - target：目标对象
    - property：引用的目标对象上的字符串键属性
- 捕获器不变式
    - 如果自有的 target.property 存在且不可配置，则处理程序必须返回一个表示该属性存在的对象
    - 如果自有的 target.property 存在且可配置，则处理程序必须返回表示该属性可配置的对象
    - 如果自有的 target.property 存在且 target 不可扩展，则处理程序必须返回一个表示该属性存在的对。
    - 如果 target.property 不存在且 target 不可扩展，则处理程序必须返回 undefined 表示该属性不存在
    - 如果 target.property 不存在，则处理程序不能返回表示该属性可配置的对象

deleteProperty()

delete操作符中被调用
Reflect.deleteProperty()

```
const myTarget = {};
const proxy = new Proxy(myTarget, {
 deleteProperty(target, property) {
 console.log('deleteProperty()');
 return Reflect.deleteProperty(...arguments)
 }
});
delete proxy.foo
// deleteProperty()
```

- 返回值
  布尔值，表示删除属性是否成功。返回非布尔值会被转型为布尔值
- 拦截的操作
    - delete proxy.property
    - delete proxy[property]
    - Reflect.deleteProperty(proxy, property)
- 捕获器处理程序参数
    - target：目标对象
    - property：引用的目标对象上的字符串键属性
- 捕获器不变式
  如果自有的 target.property 存在且不可配置，则处理程序不能删除这个属

ownKeys()

Object.keys()
Reflect.keys()

```
const myTarget = {};
const proxy = new Proxy(myTarget, {
 ownKeys(target) {
 console.log('ownKeys()');
 return Reflect.ownKeys(...arguments)
 }
});
```

```
Object.keys(proxy);
// ownKeys()
```

- 返回值
  返回包含字符串或符号的可枚举对象
- 拦截的操作
  - Object.getOwnPropertyNames(proxy)
  - Object.getOwnPropertySymbols(proxy)
  - Object.keys(proxy)
  - Reflect.ownKeys(proxy)
- 捕获器处理程序参数
  - target：目标对象
- 捕获器不变式
  返回的可枚举对象必须包含 target 的所有不可配置的自有属性
  如果 target 不可扩展，则返回可枚举对象必须准确地包含自有属性键

getPrototypeOf()

Object.getPrototypeOf()
Reflect.getPrototypeOf()

```
const myTarget = {};
const proxy = new Proxy(myTarget, {
 getPrototypeOf(target) {
 console.log('getPrototypeOf()');
 return Reflect.getPrototypeOf(...arguments)
 }
});
Object.getPrototypeOf(proxy);
// getPrototypeOf()
```

- 返回值
  返回对象或 null。
- 拦截的操作
  - Object.getPrototypeOf(proxy)
  - Reflect.getPrototypeOf(proxy)
  - proxy.**proto**
  - Object.prototype.isPrototypeOf(proxy)
  - proxy instanceof Object
- 捕获器处理程序参数
  - target：目标对象
- 捕获器不变式
  如果 target 不可扩展，则 Object.getPrototypeOf(proxy)唯一有效的返回值就是 Object

setPrototypeOf()

Object.setPrototypeOf()
Reflect.setPrototypeOf()

```
const myTarget = {};
const proxy = new Proxy(myTarget, {
 setPrototypeOf(target, prototype) {
 console.log('setPrototypeOf()');
 return Reflect.setPrototypeOf(...arguments)
 }
});
Object.setPrototypeOf(proxy, Object);
// setPrototypeOf()
```

- 返回值
  返回布尔值，表示原型赋值是否成功。返回非布尔值会被转型为布尔值
- 拦截的操作
  - Object.setPrototypeOf(proxy)
  - Reflect.setPrototypeOf(proxy)
- 捕获器处理程序参数
  - target：目标对象。
  - prototype：target 的替代原型，如果是顶级原型则为 null
- 捕获器不变式
  如果 target 不可扩展，则唯一有效的 prototype 参数就是 Object.getPrototypeOf(target)的返回值

isExtensible()

Object.isExtensible()
Reflect.isExtensible()

```
const myTarget = {};
const proxy = new Proxy(myTarget, {
 isExtensible(target) {
 console.log('isExtensible()');
 return Reflect.isExtensible(...arguments)
 }
});
Object.isExtensible(proxy);
// isExtensible()
```

- 返回值
  返回布尔值，表示 target 是否可扩展。返回非布尔值会被转型为布尔值
- 拦截的操作
  - Object.isExtensible(proxy)
  - Reflect.isExtensible(proxy)
- 捕获器处理程序参数
  - target：目标对象
- 捕获器不变式
  如果 target 可扩展，则处理程序必须返回 true
  如果 target 不可扩展，则处理程序必须返回 false

preventExtensions()

Object.preventExtensions()
Reflect.preventExtensions()

```
const myTarget = {};
const proxy = new Proxy(myTarget, {
 preventExtensions(target) {
 console.log('preventExtensions()');
 return Reflect.preventExtensions(...arguments)
 }
});
Object.preventExtensions(proxy);
// preventExtensions()
```

- 返回值
  返回布尔值，表示 target 是否已经不可扩展。返回非布尔值会被转型为布尔值。
- 拦截的操作
  - Object.preventExtensions(proxy)
  - Reflect.preventExtensions(proxy)
- 捕获器处理程序参数
  - target：目标对象
- 捕获器不变式
  如果 Object.isExtensible(proxy)是false，则处理程序必须返回 true

apply()

调用函数时被调用
Reflect.apply()

```
const myTarget = () => {};
const proxy = new Proxy(myTarget, {
 apply(target, thisArg, ...argumentsList) {
 console.log('apply()');
 return Reflect.apply(...arguments)
 }
});
proxy();
// apply()
```

- 返回值
  无限制
- 拦截的操作
  - proxy(...argumentsList)
  - Function.prototype.apply(thisArg, argumentsList)
  - Function.prototype.call(thisArg, ...argumentsList)
  - Reflect.apply(target, thisArgument, argumentsList)
- 捕获器处理程序参数
  - target：目标对象。
  - thisArg：调用函数时的 this 参数。
  - argumentsList：调用函数时的参数列表

- 捕获器不变式
  target 必须是一个函数对象

construct()

new操作符时被调用
Reflect.construct()

```
const myTarget = function() {};
const proxy = new Proxy(myTarget, {
 construct(target, argumentsList, newTarget) {
 console.log('construct()');
 return Reflect.construct(...arguments)
 }
});
new proxy;
// construct()
```

- 返回值
  返回一个对象。
- 拦截的操作
  - new proxy(...argumentsList)
  - Reflect.construct(target, argumentsList, newTarget)
- 捕获器处理程序参数
  - target：目标构造函数
  - argumentsList：传给目标构造函数的参数列表
  - newTarget：最初被调用的构造函数
- 捕获器不变式
  target 必须可以用作构造函数