Proxy | Reflect | Class

Proxy

监听对象

Es6 以前

通过 Object.defineProperty 的 get 和 set 函数

1 、新增属性、删除属性，那么 Object.defineProperty是无能为力的
2 、访问器描述符设计的初衷并不是为了去监听一个完整的对象，用在实现监听对象操作上，属于比较勉强
3 、达成监听目的是同时，将初衷原本是定义普通的属性，强行将它变成了数据属性描述符

```
const obj = { name:"xiaoyu", age:20}

//1.创建一个Proxy对象，Proxy对象是一个类，所以使用new创建
const objProxy = new Proxy(obj,{//代理obj
  set:function(target,key,value){
    console.log(`监听：监听${key}的设置值`);
    target[key] = value
  },

  get:function(target,key){
    return target[key]
  },

  deleteProperty:function(target,key){
    delete target[key]
  },
  has:function(target,key){
    return key in target//返回结果
  }
})

delete objProxy.name

console.log("age" in objProxy);//通过in 判断"age"有没有再objProxy里面，true
```

Es6

1 、希望监听一个对象的相关操作，那么我们可以先创建一个代理对象（Proxy对象），
2 、之后对该对象的所有操作，都通过代理对象来完成，代理对象可以监听我们想要对原对象进行哪些操作

拦截机制：JS引擎会在内部为Proxy对象维护一个关联的目标对象和处理器对象。当对Proxy对象进行操作时，这些操作首先被送到处理器对象
方法查找与执行：对于每种可以拦截的操作，如get、set、apply等，处理器对象可以提供一个同名的方法来拦截相应的操作，在处理器对象中查找到对应方法进行执行

配合 Proxy

Reflect进行操作肯定是有好处，例如返回值失败情况下明确false而非抛出异常，这是更可预测的错误处理方式，也不需要使用try-catch来捕获错误，更加动态灵活，更加函数式编程(Reflect方法全是函数)

receiver参数

如果我们的源对象（obj）有setter、getter的访问器属性，那么可以通过receiver来改变里面的this

```
const obj = {
  _name: 'coderwhy',
  get name() {
    return this._name
  },
  set name(newValue) {
    this._name = newValue
  }
}
const objProxy = new Proxy(obj, {
  get: function(target, key, receiver) {
    console.log('被访问：', target, key)
    return Reflect.get(target, key, receiver)
  },
  set: function(target, key, newValue, receiver) {
    console.log('被设置：', target, key)
    Reflect.set(target, key, newValue, receiver)
  }
})
// 这在Reflect.set触发之前打印的，所以输出的_name为未修改状态，在浏览器控制台则为最终结果'小余'
objProxy.name = '小余'
// 被设置： { _name: 'coderwhy', name: [Getter/Setter] } name
// 被设置： { _name: 'coderwhy', name: [Getter/Setter] } _name
console.log(objProxy.name)
// 被访问： { _name: '小余', name: [Getter/Setter] } name
// 被访问： { _name: '小余', name: [Getter/Setter] } _name
```

第一次拦截中，是正常在Proxy调用了Reflect的set与get此时的key是指obj中的name方法(setter、getter)

第二次拦截中，监听层的this被Reflect的receiver所改变，变为Proxy代理本身，此时在obj中的代码就会变为如下形式

```
const obj = {
  _name: 'coderwhy',
  get name() {
    return objProxy._name//{ _name: 'coderwhy', name: [Getter/Setter] }._name
  },
  set name(newValue) {
    objProxy._name = newValue
  }
}
```

```
const obj = {
  _name: 'test',
  get name(){
    return this._name;
  }
};
const proxyObj = new Proxy(obj,{
  get(target, property, receiver){
    return Reflect.get(target, property, receiver)
  },
});

const child = {
  _name: 'child'
};
Reflect.setPrototypeOf(child, proxyObj);

child.name // child
proxyObj.name // test
```

```
const obj = {
  _name: 'test',
  get name() {
    return this._name
  }
}
const proxyObj = new Proxy(obj, {
  get(target, property, receiver) {
    console.log('target', target, receiver)
    return target[property]
  }
})

const child = {
  _name: 'child'
}
Reflect.setPrototypeOf(child, proxyObj)

console.log(child.name) // test
console.log(proxyObj.name) // test
```

1 、Proxy.get方法的Receiver参数是：Proxy自身代理
2 、Reflect.get方法是Receiver参数是：如果target对象中指定了getter，receiver则为getter调用时的this值

Reflect

不是类，也不是构造函数或者函数对象，而是一个标准的内置对象

1 、早期的ECMA规范中没有考虑到这种对 对象本身 的操作如何设计会更加规范，所以将这些API放到了Object上面
2、对于最顶层的Object来说，身为所有类的父类，他本身不应该包含太多的东西的，因为父类里的东西是会被继承到子类中的，太多的东西必然会加重子类的负担而过于臃肿

类

创建

```
class Person{
  //类的构造方法(在这里的constructor构造函数就是一种特殊的构造方法)
  constructor(name){ }
}
var p = new Person('小余')
```

new

1、在内存中创建一个新的对象（空对象）
2、这个对象内部的[[prototype]]属性会被赋值为该类的prototype属性
3、构造函数内部的this，会指向创建出来的新对象
4、执行构造函数的内部代码（函数体代码）
5、如果构造函数没有返回非空对象，则返回创建出来的新对象

实例方法

静态方法 static

类不能访问实例方法，因为构造函数的__proto__不指向类的protoType对象，指向的是Function.protoType

访问器方法

```
var obj = {
  _name:"小余",
  //setter方法(私有属性)，这就是个名词
  set name(value){
    this._name = value
  },
  //getter方法
  get name(){
    return this._name
  }
}
```

```
class Person {
 constructor(name,age){
   this.name = name
   this.age = age
 }
}

// 使用extends实现继承 Student(前者)继承自Person(后者)
class Student extends Person {
}
```

extends

虽然利用extends将Person与Student之间形成了继承关系但我们要如何在Student中拿到Person中的这些内容呢

继承

super 关键字在类的继承中起重要作用。主要用于调用父类的构造函数、方法和访问父类的属性。从而做到子类能够重用和扩展父类的功能

在子（派生）类的构造函数中使用this或者返回默认对象之前，必须先通过super调用父类的构造函数

super

```
class Person {
  constructor(name, age) {
    this.name = name
    this.age = age
  }
  static running(){
    console.log('逻辑1');
  }
}

// 使用extends实现继承
class Student extends Person {
  constructor(name,age) {
    super(name, age)
  }
//重写静态方法
  static running1(){
    //父类方法融为子类方法的一部分
    super.running()
    console.log('逻辑4');

  }
}
```

重写

是在子类的重写方法中使用super关键字调用父类的对应方法，这个方法包含了静态方法和实例方法

```
Student.running1()//逻辑1-6
```

```
class HYArray extends Array{
  get lastItem(){
    //获取数组最后一个的数据
    return this[this.length -1]
  }

  //获取数组的第一个数据
  get firstItem(){
    return this[0]
  }
}

var arr = new HYArray(10,20,30)

console.log(arr.lastItem);//30
console.log(arr.firstItem);//10

//我们以前的做法，是直接在原型链上面进行扩展
Array.prototype.lastItem = function(){
  return this[this.length -1]
}
```

扩展

混入

类的混入（mixin）是一种在面向对象编程中用于实现代码复用的技术，它将一个类的方法和属性注入到另一个类中，从而达到功能组合和代码重用的目的，所以在这里的混入mixin是一种思想体现，而非新的语法

单继承

多继承

js依靠原形链，难以委会，性能开销

```
function calcArea(foo) {
  console.log(foo.getArea());
}

var obj1 = {
  name:'xiaoyu',
  getArea: function(){
    return 1000
  }
}

class Person {
  getArea() {
    return 100
  }
}

var p = new Person()
//实现多态
calcArea(p)
calcArea(obj1)
```

多态

不同的数据类型进行同一个操作，表现出不同的行为，就是多态的体现