【前端面试】你对JS中的Proxy代理了解多...

前言

就目前而言，前端框架基本上被 Vue 和 React 瓜分得差不多了。如果你去面试一个前端岗位，那么或多或少都会问你一些关于 Vue 和 React 框架得知识，无论是原理还是使用，我们都有必要去了解一番。

数据响应式可以说是这些框架的一大特色与核心，这里我们就拿 Vue 来说。在 Vue2.x 时代，实现数据响应式主要是使用 Object.defineProperty()这个 API 来实现的，而到了 Vue3.x 时代，数据响应式主要是使用 Proxy()来实现的。

如果你还不了解 Proxy，那么你很有必要跟着本篇文章学习一下！

1.基本概念

想要学习一个新的 API 或者知识，不能一上来就看它怎么使用。我们要学习从基本概念入手，这样才能做到有始有终。

Proxy 是在 ES6 中才被标准化的，而 Vue2.x 版本是基于 ES6 版本之前的 Object.defineProperty()设计的，我们先来看下官方是怎么解释 Proxy 的。

官网解释：

> Proxy 对象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）。

为了方便大家好理解，这里先抓几个关键词出来：

- 对象
- 创建对象代理
- 拦截
- 自定义

从上面的关键词大家应该能够揣摩个一二了，首先 Proxy 是一个对象，它可以给另外一个对象创建一个代理，代理可以简单理解为代理某一个对象，然后通过这个代理对象，可以针对于该对象提前做一些操作，比如拦截等。

通俗的解释：

> 假如我们有一个对象 obj，使用 Proxy 对象可以给我们创建一个代理，就好比我们打官司之前，可以先去律师事务所找一个律师，律师全权代理我们。有了这个代理之后，就可以对这个 obj 做一些拦截或自定义，比如对方想要直接找我们谈话时，我们的律师可以先进行拦截，他来判断是否允许和我谈话，然后再做决定。这个律师就是我们对象的代理，有人想要修改 obj 对象，必须先经过律师那一关。

基本概念其实不复杂，有些小伙伴不太理解的原因大多是平时写代码的时候，以为对象就是一个独立的变量，比如声明了一个对象 obj={name:”小猪课堂”}，我们通常也不会去做什么拦截，想改就改。

这就是一个惯性思维！

2.如何使用

既然我们知道了 Proxy 的作用，那么我们如何使用它呢？或者说如何给一个对象创建代理。

Proxy 的使用非常简单，我们可以使用 new 关键字实例化它。

代码如下：

```
const p = new Proxy(target, handler)
```

代码非常的简单，重点是我们需要掌握 Proxy 接收的参数。

参数说明

target:

需要被代理的对象，它可以是任何类型的对象，比如数组、函数等等，注意不能是基础数据类型。

示例代码：

```
<script>
  let obj = {
    name: '小猪课堂',
    age: 23
  }
  let p = new Proxy(obj, handler);
</script>
```

handler:

它是一个对象，该对象的属性通常都是一些函数，handler 对象中的这些函数也就是我们的处理器函数，主要定义我们在代理对象后的拦截或者自定义的行为。handler 对象的的属性大概有下面这些，具体使用方法我们在后面章节详解：

- handler.apply()
- handler.construct()

- handler.defineProperty()
- handler.deleteProperty()
- handler.get()
- handler.getOwnPropertyDescriptor()
- handler.getPrototypeOf()
- handler.has()
- handler.isExtensible()
- handler.ownKeys()
- handler.preventExtensions()
- handler.set()
- handler.setPrototypeOf()

我们使用 new 关键词后生成了一个代理对象 p，它就和我们原来的对象 obj 一样，只不过它是一个 Proxy 对象，我们打印出来看看就能更好理解了。

示例代码：

```
<script>
  let obj = {
    name: '小猪课堂',
    age: 23
  }
  let p = new Proxy(obj, {});
  console.log(obj);
  console.log(p);
</script>
```

输出结果：

3.Handler 对象详解

上一节的使用只是简单的初始化了一个代理对象，而我们需要重点掌握的是 Proxy 对象中的 handler 参数。因为我们所有的拦截操作都是通过这个对象里面的函数而完成的。

就好比律师全权代理了我们，那他拦截之后能做什么呢？或者说律师拦截之后他有哪些能力呢？这就是我们 handler 参数对象的作用了，接下来我们就一一来讲解下。

3.1 handler.apply

该方法主要用于函数调用的拦截，比如我们代理的对象是一个函数，那么我们代理这个函数之后，可以在它调用之前做一些我们想做的事。

语法：

```
// 函数拦截
let p1 = new Proxy(target, {
  apply: function (target, thisArg, argumentsList) {
  }
});
```

参数解释：

- target：被代理对象，也就是目标函数
- thisArg：调用时的上下文对象，也就是 this 指向，它绑定在 handler 对象上面
- argumentsList：函数调用的参数数组

使用案例：

```
function sum(a, b) {
  return a + b;
}
let p1 = new Proxy(sum, {
  apply: function (target, thisArg, argumentsList) {
    return argumentsList[0] + argumentsList[1] * 100;
  }
});
// 正常调用
console.log(sum(1, 2)); // 3
// 代理之后调用
console.log(p1(1, 2)); // 201
```

上段代码中我们代理了 sum 函数对象，并产生了新的 p1 代理对象，在 p1 代理对象里面，我们对函数的调用做了拦截，让它返回了新的值。

注意：

我们这里代理的函数对象必须是可调用的，也就是 target 可调用，否则会报错。

3.2 handler.construct

该方法主要是用于拦截 new 操作符的，我们通常使用 new 操作符都是在函数的情况下，但是我们不能说 new 操作符只能作用与函数，确切的说 new 操作符必须作用于自身带有

[[Construct]]内部方法的对象上，而这种对象通常就是函数，总之一句话，使用 new targe 是必须有效的。

语法：

```
// 构造函数拦截
let p2 = new Proxy(target, {
  construct: function (target, argumentsList, newTarget) {
  }
});
```

参数解释：

- target：被代理对象，需要能够使用 new 操作符初始化它的实例，通常就是一个函数
- argumentsList：使用 new 操作符是传入的参数列表
- newTarget：被调用的构造函数，也就是 p2

使用案例：

```
let p2 = new Proxy(function () { }, {
  construct: function (target, argumentsList, newTarget) {
    return { value: '我是' + argumentsList[0] };
  }
});
console.log(new p2("小猪课堂")); // {value: '我是小猪课堂'}
```

上段代码中 p2 就是一个构造函数，只不过是代理之后的新函数，我们使用 new 操作符实例化它的，首先就会去执行 handler 里面的 construct 方法。

注意：

这里有两个点需要大家注意

- target 必须能够使用 new 操作符初始化
- construct 必须返回一个对象

3.3 handler.defineProperty

这个方法其实比较有意思，Object.defineProperty 方法本身就有拦截对象的意思在里面，但是我们的 Proxy 对象可以正针对 Object.defineProperty 操作进行拦截，对于 Object.defineProperty 方法不熟悉的同学可以先去学学。

语法：

```
// 拦截 Object.defineProperty
let p3 = new Proxy(target, {
  defineProperty: function (target, property, descriptor) {
  }
});
```

参数解释：

- target：被代理对象
- property：属性名，也就是当我们使用 Object.defineProperty 操作的对象的某个属性
- descriptor：待定义或修改的属性的描述符

使用案例：

```
let p3 = new Proxy({}, {
  defineProperty: function (target, property, descriptor) {
    descriptor.enumerable = false; // 修改属性描述符
    console.log(property, descriptor);
    return true;
  }
});
let desc = { configurable: true, enumerable: true, value: 10 };
Object.defineProperty(p3, 'a', desc); // a {value: 10, enumerable: false, co
```

上段代码中我们使用 Proxy 代理了一个空对象，并产生了新的代理对象 p3，当使用 Object.defineProperty 操作 p3 对象时，就会触发 handler 中的 defineProperty 方法。

注意：

- 被代理的对象必须要能被扩展
- hanlder 中的 defineProperty 方法必须返回一个 Boolean 值
- 不能添加或者修改一个属性为不可配置的，如果它不作为一个目标对象的不可配置的属性存在的话

3.4 handler.deleteProperty

该方法用于拦截对对象属性的 delete 操作，我们经常使用 delete 删除对象中的某个属性，我们可以使用 deleteProperty 方法对该做进行拦截。

语法：

```
let p4 = new Proxy(target, {
  deleteProperty: function (target, property) {
  }
});
```

参数解释：

- target：被代理的目标对象
- property：将要被删除的属性

使用案例：

```
let p4 = new Proxy({}, {
  deleteProperty: function (target, property) {
    console.log("将要删除属性：", property)
  }
});
```

```
delete p4.a; // 将要删除属性：a
```

当我们删除 p4 对象的属性时，便会执行 handler 中的 deleteProperty 方法。

注意：

代理的目标对象的属性必须是可配置的，即可以删除，否则会报错。

3.5 handler.get

该方法用于拦截对象的读取属性操作，比如我们要读取某个对象的属性，就可以使用该方法进行拦截。

语法：

```
// 拦截读取属性操作
let p5 = new Proxy(target, {
  get: function (target, property, receiver) {
  }
});
```

参数解释：

- target：被代理的目标对象
- property：想要获取的属性名
- receiver：Proxy 或者继承 Proxy 的对象

使用案例：

```
// 拦截读取属性操作
let p5 = new Proxy({}, {
  get: function (target, property, receiver) {
    console.log("属性名：", property); // 属性名：name
    console.log(receiver); // Proxy {}
    return '小猪课堂'
  }
});
console.log(p5.name); // 小猪课堂
```

可以看到我们代理的对象其实是一个空对象，但是我们获取 name 属性是是返回了值的，其实是在 handler 对象中的 get 函数返回的。

注意：

代理的对象属性必须是可配置的，get 函数可以返回任意值。

3.6 handler.getOwnPropertyDescriptor

该方法用于拦截 Object.getOwnPropertyDescriptor 操作，也可以说它是该方法的钩子，如果对 getOwnPropertyDescriptor 还不熟悉的小伙伴可以先去了解一下。

语法：

```
let p6 = new Proxy(target, {
  getOwnPropertyDescriptor: function (target, prop) {
  }
});
```

参数解释：

- target：被代理的目标对象
- prop：返回属性名称的描述

使用案例：

```
let p6 = new Proxy({ name: '小猪课堂' }, {
  getOwnPropertyDescriptor: function (target, prop) {
    console.log('属性名称：' + prop); // 属性名称：name
    return { configurable: true, enumerable: true, value: '张三' };
  }
});
console.log(Object.getOwnPropertyDescriptor(p6, 'name').value); // 张三
```

上段代码中我们在拦截其中重新设置了属性描述，所以最后打印的 value 是”张三“。

注意：

- getOwnPropertyDescriptor 必须返回一个 object 或 undefined。
- 使用 getOwnPropertyDescriptor 时，目标对象的该属性必须存在

3.7 handler.getPrototypeOf

当我们读取代理对象的原型时，会触发 handler 中的 etPrototypeOf 方法。

语法：

```
let p7 = new Proxy(obj, {
  getPrototypeOf(target) {
  }
});
```

参数解释：

- target：被代理的目标对象

使用案例：

```
let p7 = new Proxy({}, {
  getPrototypeOf(target) {
    return { msg: "拦截获取对象原型操作" }
  }
});
console.log(p7.__proto__); // {msg: '拦截获取对象原型操作'}
```

以下操作会触发代理对象的该拦截方法：

- Object.getPrototypeOf()

- Reflect.getPrototypeOf()
- __proto__
- Object.prototype.isPrototypeOf()
- instanceof

注意：

getPrototypeOf 方法必须返回一个对象或者 null。

3.8 handler.has

该拦截方法主要是针对 in 操作符的，in 操作符通常用来检测某个属性是否存在某个对象内。

语法：

```
let p8 = new Proxy(target, {
  has: function (target, prop) {
  }
});
```

参数解释：

- target：被代理的目标对象
- prop：需要检查是否存在的属性

以下操作可以触发该拦截函数：

- 属性查询：foo in proxy
- 继承属性查询：foo in Object.create(proxy)
- with 检查: with(proxy) { (foo); }
- Reflect.has()

使用案例：

```
let p8 = new Proxy({}, {
  has: function (target, prop) {
    console.log('检测的属性: ' + prop); // 检测的属性: a
    return true;
  }
});
console.log('a' in p8); // true
```

上段代码中我们代理对象是其实没有 a 属性，但是我们拦截之后直接返回的一个 true。

注意：

has 函数返回的必须是一个 Boolean 值。

3.9 handler.isExtensible

Object.isExtensible()方法主要是用来判断一个对象是否可以扩展，handler 中的 isExtensible 方法可以拦截该操作。

语法：

```
// 拦截 Object.isExtensible()
let p9 = new Proxy(target, {
  isExtensible: function (target) {
  }
});
```

参数解释：

- target：被代理的目标对象

使用案例：

```
let p9 = new Proxy({}, {
  isExtensible: function (target) {
    console.log('操作被拦截了');
    return true;
  }
});
console.log(Object.isExtensible(p9));
```

注意：

isExtensible 方法必须返回一个 Boolean 值或可转换成 Boolean 的值。

3.10 handler.ownKeys

静态方法 Reflect.ownKeys() 返回一个由目标对象自身的属性键组成的数组。handler 对象的 ownKeys 方法可以拦截该操作，除此之外，还有一些其它操作也会触发 ownKeys 操作。

语法：

```
let p10 = new Proxy(target, {
  ownKeys: function (target) {
  }
});
```

参数解释：

- target：被代理的目标对象

以下操作会触发拦截：

- Object.getOwnPropertyNames()
- Object.getOwnPropertySymbols()
- Object.keys()
- Reflect.ownKeys()

使用案例：

```
let p10 = new Proxy({}, {
  ownKeys: function (target) {
    console.log('被拦截了');
    return ['a', 'b', 'c'];
  }
});
console.log(Object.getOwnPropertyNames(p10)); // ['a', 'b', 'c']
```

注意：

ownKeys 的结果必须是一个数组，数组的元素类型要么是一个 String，要么是一个 Symbol。

3.11 handler.preventExtensions

Object.preventExtensions()方法让一个对象变的不可扩展，也就是永远不能再添加新的属性。

handler.preventExtensions 可以拦截该项操作。

语法：

```
let p11 = new Proxy(target, {
  preventExtensions: function (target) {
  }
});
```

参数解释：

- target：被代理的目标对象

使用案例：

```
let p11 = new Proxy({}, {
  preventExtensions: function (target) {
    console.log('被拦截了');
    Object.preventExtensions(target);
    return true
  }
});

Object.preventExtensions(p11);
```

以下操作会触发拦截：

- Object.preventExtensions()
- Reflect.preventExtensions()

注意：

如果目标对象是可扩展的，那么只能返回 false

3.12 handler.set

当我们给对象设置属性值时，将会触发该拦截。

语法：

```
let p12 = new Proxy(target, {
  set: function (target, property, value, receiver) {
  }
});
```

参数解释：

- target：被代理的目标对象
- property：将要被设置的属性名
- value：新的属性值
- receiver：最初被调用的对象，通常就是 proxy 对象本身

以下操作会触发拦截：

- 指定属性值：proxy[foo] = bar 和 proxy.foo = bar
- 指定继承者的属性值：Object.create(proxy)[foo] = bar
- Reflect.set()

使用案例：

```
let p12 = new Proxy({}, {
  set: function (target, property, value, receiver) {
    target[property] = value;
    console.log('property set: ' + property + ' = ' + value); // property se
    return true;
  }
});
p12.a = 10;
```

注意：

set() 方法应当返回一个布尔值

3.13 handler.setPrototypeOf

Object.setPrototypeOf() 方法设置一个指定的对象的原型，当调用该方法修改对象的原型时就会触发该拦截。

语法：

```
let p13 = new Proxy(target, {
  setPrototypeOf: function (target, prototype) {
  }
});
```

参数解释：

- target：被代理的目标对象
- prototype：对象新原型或者为 null

使用案例：

```
let p13 = new Proxy({}, {
  setPrototypeOf: function (target, prototype) {
    console.log("触发拦截"); // 触发拦截
    return true;
  }
});
Object.setPrototypeOf(p13, {name: '小猪课堂'})
```

注意：

如果成功修改了[[Prototype]], setPrototypeOf 方法返回 true,否则返回 false。

总结

很多小伙伴可能因为 Vue3 的原因知道了 Proxy 代理的存在，但是很多都只了解 set、get 等方法，其实 Proxy 提供了很多拦截供我们使用，具体在什么场景下使用什么拦截函数，还需要自己独立思考。