【前端面试】instanceof实现的原理是什么?

前言

类型的判断可以说在我们前端开发过程中无处不在，特别是在 Typescript 还未推出之前，我们在 JS 里面做类型判断显得就更加重要了。

判断数据类型的方式有特别多，比如大家常用的 typeof、instanceof、Object.prototype.tostring.call()等等，那么每一种判断数据类的方法大家知道其中的原理吗?

比如说 instanceof 的原理，今天我们就聊一聊 instanceof 是如何判断数据类型的。

1.基本概念

想要了解 instanceof 的原理，我们至少应该知道它的基本概念吧，我们可以先来看看官网是如何解释它的。

官网解释：

> instanceof 运算符用于检测构造函数的 prototype 属性是否出现在某个实例对象的原型链上。

虽然官方的解释只有简短的一句话，但是对于许多人来说还是挺难理解的，不过我们可以抓出这句话中的几个关键点：

- **运算符**
- **构造函数的 prototype**
- **对象原型链**

很明显，instanceof 与原型和原型链有关，所以强烈建议小伙伴们先去学一学原型和原型链的相关知识。

为了让小伙伴现有一个大概理解，我们用我们自己的话简单说一下 instance。

通俗的解释：

> instanceof 是一个运算符，它可以用来判断某一个对象的类型，具体原理就是利用了原型和原型链。

基本用法：

```
A instanceof B  // true or false
```

上段代码中的 A 就是我们需要判断类型的对象，B 就是官方所说的构造函数，形如我们的 Object、Function 都可以称之为构造函数。

2.与 typeof 对比

我们判断类型的时候通常是将 typeof 和 instanceof 结合使用，虽然它们都可以判断数据类型，但是它们还是有很多不同点的，如下：

- typeof：主要用来判断基础数据类型，比如：Number、String 等等。
- instanceof：主要用来判断对象数据类型，比如 Function、Array 等等。
- typeof 直接返回数据类型，而 instanceof 重在判断，它返回布尔值。

我们来看一段代码大家可能会更好理解一些。

代码如下：

```
<script>
  function say() { };

  // typeof 判断数据类型
  console.log(typeof '小猪课堂'); // string
  console.log(typeof 100); // number
  console.log(typeof true); // boolean
  console.log(typeof undefined); // undefined
  console.log(typeof {}); // object
  console.log(typeof []); // object
  console.log(typeof null); // object
  console.log(typeof say); // function

  // instanceof 判断数据类型
  let a = new String('123');
  let b = new say();
  console.log('123' instanceof String); // false
  console.log(a instanceof String); // true
  console.log([] instanceof Array); // true
  console.log(b instanceof say); // true
</script>
```

从上段代码我们可以看出 typeof 只能判断基础数据类型（null 除外），当判断其它数据类型时，它总是返回 object 或者 function。

而 instanceof 可以用来判断对象数据类型，返回的是布尔值。

3.instanceof 特点

上节中有一段代码我们可以拿出来再看一看：

```
let b = new say();
console.log(b instanceof say); // true
```

上段代码中 b 是实例对象，say 是构造函数，我们利用 instanceof 来进行判断时，返回的 true，由此我们可以总结出 instanceof 如下特点：

- instanceof 左侧是一个实例对象，右侧是一个构造函数。
- 如果实例对象属于构造函数，那么 instanceof 就会返回 true。

我们判断类型是使用的 Array、Object、String 等等其实就是一个构造函数。

总结：

由上可以得出，判断数据类型并不是 instanceof 最准确的说法，它主要是用来判断实例对象与构造函数之间的关系的。而判断数据类型只是我们利用它的特点变相实现罢了。

4.instanceof 原理

到这里我们知道 instanceof 其实不仅仅是用来判断数据类型的，它实际上是用来判断一个实例对象与一个构造函数之间的关系的。

那么我们通常如何判断一个实例对象与一个构造函数之间的关系的呢?

答案就是利用**原型和原型链！**我们都知道每一个函数都有一个显式原型 prototype，每一个对象都有一个隐式原型__proto__，当我们对象的原型链中存在构造函数的显式原型 prototype 时，我们就可以确定它们之间时存在关系的。

更简单的说法：

- 我们拿到 instanceof 左侧对象的原型链
- 再拿到 instanceof 右侧构造函数的显式原型 prototype
- 如果原型链中存在显式原型 prototype，instanceof 返回 true，否则返回 false

如果大家对上面的说明看的模糊，那么快去补一补原型和原型链的知识。

我们可以简单实现一个 instanceof 函数，大家就更容易理解了。

代码如下：

```
/**
  * @description 判断对象是否属于某个构造函数
  * @prams left: 实例对象  right: 构造函数
  * @return boolean
*/
function myInstanceof(left, right) {
  let rightPrototype = right.prototype; // 获取构造函数的显式原型
  let leftProto = left.__proto__; // 获取实例对象的隐式原型
  while (true) {
    // 说明到原型链顶端，还未找到，返回 false
    if (leftProto === null) {
      return false;
    }
    // 隐式原型与显式原型相等
    if (leftProto === rightPrototype) {
```

```
      return true;
    }
    // 获取隐式原型的隐式原型，重新赋值给 leftProto
    leftProto = leftProto.__proto__
  }
}
```

代码比较简单，主要就是需要循环实例对象的原型链。

我们回过头再看一遍代码：

```
let b = new say();
console.log(b instanceof say); // true
```

上段代码为什么会返回 true 呢，其实是因为在 new 的操作过程中，有一步操作便是将 say()构造函数的显式原型 prototype 赋值给了 b 的隐式原型__proto__，所以我们利用 instance 判断时，必然会满足 leftProto === rightPrototype 条件。

至于 new 操作符具体做了什么，大家可以去参考我的另一篇文章。

大家也可以直接打印看看结果：

```
console.log(b.__proto__ === say.prototype); // true
```

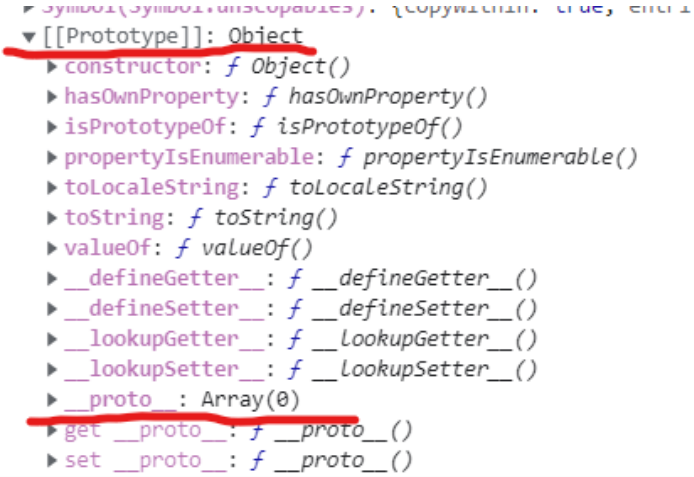
5.补充

instanceof 判断数组时，如果把它归纳为 Array 是返回 true，如果把它归纳为 Object 也是返回 true 的。

代码如下：

```
let array = [1, 2, 3]
console.log(array instanceof Array); // true
console.log(array instanceof Object); // true
```

究其原因其实是我们的数组也是一个对象，只不过这个对象稍微特殊一点罢了，大家也可以把数组的原型打印出来看看，一下就会明白了。



总结

看到这儿，你在回过头去看看官网关于 instanceof 的解释，相信你会恍然大悟！

> instanceof 运算符用于检测构造函数的 prototype 属性是否出现在某个实例对象的原型链上。