面试官:请你实现一个深克隆

如何实现一个深克隆

前言

实现一个深克隆是面试中常见的问题的,可是绝大多数面试者的答案都是不完整的,甚至是错误的,这个时候面试官会不断追问,看看你到底理解不理解深克隆的原理,很多情况下一些一知半解的面试者就原形毕漏了.

我们就来看一下如何实现一个深克隆,当然面试中没有让你完整实现的时候,但是你一定要搞清楚其中的坑在哪里,才可以轻松应对面试官的追问.

在要实现一个深克隆之前我们需要了解一下javascript中的基础类型.

javascript基础类型

> JavaScript原始类型:Undefined、Null、Boolean、Number、String、Symbol
>
> JavaScript引用类型:Object

1.浅克隆

浅克隆之所以被称为**浅克隆**，是因为对象只会被克隆最外部的一层,至于更深层的对象,则依然是通过引用指向同一块堆内存.

```
复制代码
// 浅克隆函数function shallowClone(o) {  const obj = {};  for ( let i in o) {  obj[i] = o[i];  }  return obj;}// 被克隆对象const oldObj = {  a: 1,  b: [ 'e', 'f', 'g' ],  c: { h: { i: 2 } }};const newObj = shallowClone(oldObj);console.log(newObj.c.h, oldObj.c.h); // { i: 2 } { i: 2 }console.log(oldObj.c.h === newObj.c.h); // true
```

我们可以看到,很明显虽然 `oldObj.c.h` 被克隆了,但是它还与 `oldObj.c.h` 相等,这表明他们依然指向同一段堆内存,这就造成了如果对 `newObj.c.h` 进行修改,也会影响 `oldObj.c.h` ,这就不是一版好的克隆.

```
复制代码
newObj.c.h.i = 'change';console.log(newObj.c.h, oldObj.c.h); // { i: 'change' } { i: 'change' }
```

我们改变了 `newObj.c.h.i` 的值, `oldObj.c.h.i` 也被改变了,这就是浅克隆的问题所在.

当然有一个新的api `Object.assign()` 也可以实现浅复制,但是效果跟上面没有差别,所以我们不再细说了.

2.深克隆

2.1 JSON.parse方法

前几年微博上流传着一个传说中最便捷实现深克隆的方法, JSON对象parse方法可以将JSON字符串反序列化成JS对象，stringify方法可以将JS对象序列化成JSON字符串,这两个方法结合起来就能产生一个便捷的深克隆.

```
复制代码
const newObj = JSON.parse(JSON.stringify(oldObj));
```

我们依然用上一节的例子进行测试

```
复制代码
const oldObj = {  a: 1,  b: [ 'e', 'f', 'g' ],  c: { h: { i: 2 } }};const
newObj = JSON.parse(JSON.stringify(oldObj));console.log(newObj.c.h,
oldObj.c.h); // { i: 2 } { i: 2 }console.log(oldObj.c.h === newObj.c.h); //
falsenewObj.c.h.i = 'change';console.log(newObj.c.h, oldObj.c.h); // { i:
'change' } { i: 2 }
```

果然,这是一个实现深克隆的好方法,但是这个解决办法是不是太过简单了.

确实,这个方法虽然可以解决绝大部分是使用场景,但是却有很多坑.

1.他无法实现对函数 、RegExp等特殊对象的克隆

2.会抛弃对象的constructor,所有的构造函数会指向Object

3.对象有循环引用,会报错

主要的坑就是以上几点,我们一一测试下.

```
复制代码
// 构造函数function person(pname) {  this.name = pname;}const Messi = new
person('Messi');// 函数function say() {  console.log('hi');};const oldObj = {
a: say,  b: new Array(1),  c: new RegExp('ab+c', 'i'),  d: Messi};const newObj
= JSON.parse(JSON.stringify(oldObj));// 无法复制函数console.log(newObj.a,
oldObj.a); // undefined [Function: say]// 稀疏数组复制错误
console.log(newObj.b[0], oldObj.b[0]); // null undefined// 无法复制正则对象
```

```
console.log(newObj.c, oldObj.c); // {} /ab+c/i// 构造函数指向错误
console.log(newObj.d.constructor, oldObj.d.constructor); // [Function: Object]
[Function: person]
```

我们可以看到在对函数、正则对象、稀疏数组等对象克隆时会发生意外，构造函数指向也会发生错误。

```
复制代码
const oldObj = {};oldObj.a = oldObj;const newObj =
JSON.parse(JSON.stringify(oldObj));console.log(newObj.a, oldObj.a); //
TypeError: Converting circular structure to JSON
```

对象的循环引用会抛出错误.

2.2 构造一个深克隆函数

我们知道要想实现一个靠谱的深克隆方法,上一节提到的**序列/反序列**是不可能了,而通常教程里提到的方法也是不靠谱的,他们存在的问题跟上一届序列反序列操作中凸显的问题是一致的.

```
function isArray (arr) {
    return Object.prototype.toString.call(arr) === '[object Array]';
}
// 深度克隆
function deepClone (obj) {
    if(typeof obj !== "object" && typeof obj !== 'function') {
        return obj;        //原始类型直接返回
    }
    var o = isArray(obj) ? [] : {};
    for(i in obj) {
        if(obj.hasOwnProperty(i)){
            o[i] = typeof obj[i] === "object" ? deepClone(obj[i]) : obj[i];
        }
    }
    return o;
}
```

(这个方法也会出现上一节提到的问题)

由于要面对不同的对象(正则、数组、Date等)要采用不同的处理方式，我们需要实现一个对象类型判断函数。

```
复制代码
const isType = (obj, type) => {  if (typeof obj !== 'object') return false;
const typeString = Object.prototype.toString.call(obj);  let flag;  switch
(type) {    case 'Array':      flag = typeString === '[object Array]';
```

```
    break;    case 'Date':      flag = typeString === '[object Date]';      break;
      case 'RegExp':      flag = typeString === '[object RegExp]';      break;
    default:      flag = false;  }  return flag;};
```

这样我们就可以对特殊对象进行类型判断了,从而采用针对性的克隆策略.

```
1 复制代码
2 const arr = Array.of(3, 4, 5, 2);console.log(isType(arr, 'Array')); // true
```

对于正则对象,我们在处理之前要先补充一点新知识.

我们需要通过正则的扩展了解到 `flags` 属性等等,因此我们需要实现一个提取flags的函数.

```
1 复制代码
2 const getRegExp = re => {  var flags = '';  if (re.global) flags += 'g';  if
  (re.ignoreCase) flags += 'i';  if (re.multiline) flags += 'm';  return flags;};
```

做好了这些准备工作,我们就可以进行深克隆的实现了.

- 复制代码
  /**deep clone@param {[type]} parent object 需要进行克隆的对象@return {[type]} 深克隆后的对象*/const clone = parent => { // 维护两个储存循环引用的数组 const parents = []; const children = []; const _clone = parent => { if (parent === null) return null; if (typeof parent !== 'object') return parent; let child, proto; if (isType(parent, 'Array')) { // 对数组做特殊处理 child = []; } else if (isType(parent, 'RegExp')) { // 对正则对象做特殊处理 child = new RegExp(parent.source, getRegExp(parent)); if (parent.lastIndex) child.lastIndex = parent.lastIndex; } else if (isType(parent, 'Date')) { // 对Date对象做特殊处理 child = new Date(parent.getTime()); } else { // 处理对象原型 proto = Object.getPrototypeOf(parent); // 利用Object.create切断原型链 child = Object.create(proto); } // 处理循环引用 const index = parents.indexOf(parent); if (index != -1) { // 如果父数组存在本对象,说明之前已经被引用过,直接返回此对象 return children[index]; } parents.push(parent); children.push(child); for (let i in parent) { // 递归 child[i] = _clone(parent[i]); } return child; }; return _clone(parent);};

我们做一下测试

```
1 复制代码
2 function person(pname) {  this.name = pname;}const Messi = new
  person('Messi');function say() {  console.log('hi');}const oldObj = {  a: say,
   c: new RegExp('ab+c', 'i'),  d: Messi,};oldObj.b = oldObj;const newObj =
```

```
clone(oldObj);console.log(newObj.a, oldObj.a); // [Function: say] [Function:
say]console.log(newObj.b, oldObj.b); // { a: [Function: say], c: /ab+c/i, d:
person { name: 'Messi' }, b: [Circular] } { a: [Function: say], c: /ab+c/i, d:
person { name: 'Messi' }, b: [Circular] }console.log(newObj.c, oldObj.c); //
/ab+c/i /ab+c/iconsole.log(newObj.d.constructor, oldObj.d.constructor); //
[Function: person] [Function: person]
```

当然,我们这个深克隆还不算完美,例如Buffer对象、Promise、Set、Map可能都需要我们做特殊处理，另外对于确保没有循环引用的对象，我们可以省去对循环引用的特殊处理，因为这很消耗时间，不过一个基本的深克隆函数我们已经实现了。

总结

实现一个完整的深克隆是由许多坑要踩的,npm上一些库的实现也不够完整,在生产环境中最好用 `lodash` 的深克隆实现.

在面试过程中,我们上面提到的众多坑是面试官很可能追问你的,要知道坑在哪里,能答出来才是你的加分项,在面试过程中必须要有一两个闪光点,如果只知道**序列/反序列**这种投机取巧的方法,在追问下不仅拿不到分,很可能造成只懂个皮毛的印象,毕竟,面试面得就是你知识的深度.