你真的知道什么是深度克隆么？

一说到深层克隆，大家多会跃跃欲试的表明，大家都会，但是大家知道真正的深层克隆是什么样子的么，本篇文章将会告诉你。

浅层克隆与深度克隆

浅层克隆也被称为浅克隆，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.

浅层克隆代码实现：

```
function shallowClone(o) {
    const obj = {};
    for ( let i in o) {
      obj[i] = o[i];
    }
    return obj;
  }
  // 被克隆对象
  const oldObj = {
    name: 'pwd',
    list: [ 'e', 'f', 'g' ],
    obj: { h: { i: 2 } }
  };

  const newObj = shallowClone(oldObj);
  console.log(newObj.obj.h, oldObj.obj.h); // { i: 2 }
{ i: 2 }
  console.log(oldObj.obj.h === newObj.obj.h); // true
```

我们可以看到,**很明显,虽然 oldObj.obj.h 被克隆了,但是它还与 oldObj.obj.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.obj.h 进行修改,也会影响 oldObj.obj.h,**这就不是一版好的克隆.

```
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。
在上面很明显我们想要的克隆，浅层克隆是远远不够的，我们的目标致力于：**两个长的一模一样的对象，但是彼此之间没有关联**。那么接下来开始我们的重头戏—---深层克隆

废话不多说，先上代码（大家最最常见的深层克隆代码）：

常见深层克隆

```
// 克隆函数
function deepClone(obj, newObj) {

    if (obj instanceof Array) { // 判断是否是数组
        newObj = [];
        return deepCloneArray(obj, newObj);
    } else if (obj instanceof Object) { // 判断是否是对象
        newObj = {};
        return deepCloneObject(obj, newObj);
    } else {
        return newObj = obj;
    }
}

// 克隆对象
function deepCloneObject(obj, newObj) {
    for (var temp in obj) {
        if (obj.hasOwnProperty(temp)) {  // 过滤原型属性
            if (obj[temp] instanceof Object || obj[temp]
instanceof Array) {  // 如果还是对象或者数组继续递归深层克隆
```

```
                var tempNewObj = {};
                newObj[temp] = deepClone(obj[temp],
tempNewObj);
            } else { // 不是直接赋值
                newObj[temp] = obj[temp];
            }
        }
    }
    return newObj;
}

// 克隆数组
function deepCloneArray(arr, newArr) {
    for (var i = 0; i < arr.length; i++) {
        if (arr[i] instanceof Object || arr[i] instanceof
Array) { // 如果还是对象或者数组继续递归深层克隆
            var tempNewObj;
            newArr[i] = deepClone(arr[i], tempNewObj);
        } else { // 不是直接赋值
            newArr[i] = arr[i];
        }
    }
    return newArr;
}
```

这个代码是大家见到最常见的 “克隆代码”，对于一些简单的克隆是可以的，比如下面的：

```
var obj = {
    name: "panda",
    sex: 18,
    msg: {
        a: 1,
        b: 2
    },
    list: [1,2,3,4]
}
```

```
var obj1 = deepClone(obj)
console.log(obj1.list, obj.list) // [ 1, 2, 3, 4 ] [ 1,
2, 3, 4 ]
console.log(obj1.list == obj.list) // false
```

以上的克隆对于一般数组和对象有效，但是我们的工作中对象不单纯只有普通对象把。当我们遇到函数，正则，日期对象会怎么样，看以下的问题：

1. 可以克隆函数么？
2. 可以克隆正则，Date 对象么？
3. 可以克隆原型么？
4. 可以解决循环引用么（环）

我们来测试一下

克隆函数：

```
var obj = {fn: function () {}}
var newObj = deepClone(obj)
console.log(newObj, obj)
```

结果： { fn: {} } { fn: [Function: fn] }，失败。

克隆正则

```
var obj = /abc/g
var newObj = deepClone(obj)
console.log(newObj, obj)
```

结果： {} /abc/g， 失败。

克隆 Date 对象

```
var obj = new Date()
var newObj = deepClone(obj)
```

```
console.log(newObj, obj)
```

结果： {} 2019-04-23T05:47:22.133Z 失败。

克隆对象原型

```
function Person() {

}
Person.prototype.getMsg = function () {

}
var p = new Person()

var obj = p
var newObj = deepClone(obj)
console.log(newObj.__proto__ ==  obj.__proto__)
```

结果： false 失败。

在我们的对象中出现循环引用，又会怎么样？

```
var a={"name":"zzz"};
var b={"name":"vvv"};
a.child=b;
b.parent=a;

var newObj = deepClone(b)
```

结果 ： RangeError: Maximum call stack size exceeded (爆栈了)

这是我们的深层克隆函数，虽说有诸多的问题，我们在后面进行解决。接下来我们在看一中简便的方法（江湖流传的妙招）。

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.

```
const newObj = JSON.parse(JSON.stringify(oldObj));
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); // false
newObj.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。

对于这里的问题和之前的问题差不多，我就不一一测试了，同学们可以回去玩一玩，是具有这样的问题。

说了这么多，把几种常见的深层克隆的方法个大家讲解后，遗留了一堆问题。

接下来我们要写个能解决以上问题的深层克隆。

首先我们分析一下遗留问题，其实上面的问题我们可以分为三类：

对象类型问题，

原型问题，

循环引用问题

那我们就逐个击破就好了。

1. 对象类型问题：

   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。

```
function isType (obj, type)  {
    if (typeof obj !== 'object') return false;
    var typeString = Object.prototype.toString.call(obj);
    var 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;
  };
```

通过这个函数 isType 我们可以根据 type 确定 obj 是不是对应的对象类型。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。

对于正则对象,我们在处理之前要先补充一点新知识。

我们需要通过正则语法解到 flags 属性等等,因此我们需要实现一个提取 flags 的函数。

```
function getRegExp (re) {
```

```
    var flags = '';
    if (re.global) flags += 'g';
    if (re.ignoreCase) flags += 'i';
    if (re.multiline) flags += 'm';
    return flags;
  };
```

2. 原型问题：

我们利用 Object.getPrototypeOf(); 获取对象原型，在利用 Object.create 切断原型链即可。

3. 循环引用问题：

对于引用值，我们通过数组进行记录，每次碰到引用值后，遍历数组进行判断。然后处理。

做好了这些准备工作,我们就可以进行深克隆的实现了。

全部代码：

```
function getRegExp (re) {
    var flags = '';
    if (re.global) flags += 'g';
    if (re.ignoreCase) flags += 'i';
    if (re.multiline) flags += 'm';
    return flags;
  };

function isType (obj, type)  {
    if (typeof obj !== 'object') return false;
    var typeString = Object.prototype.toString.call(obj);
    var 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;
  };


function clone (parent)  {
    // 维护两个储存循环引用的数组
    var parents = [];
    var children = [];

    var _clone = parent => {
      if (parent === null) return null;
      if (typeof parent !== 'object') return parent;

      var 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
      var 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 (var i in parent) {
        // 递归

        if(parent.hasOwnProperty(i)) {  // 过滤原型属性
            child[i] = _clone(parent[i]);
        }

      }

      return child;
    };
    return _clone(parent);
  };
```

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。

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。