Array.prototype.flat()

数组扁平化是 `JavaScript` 面试中考查率很高的内容，其核心便是递归思想

递归在很多人看来都是很难的东西，但是实际上递归问题处理起来是有技巧的

递归问题，实际上就是一个问题可以逐步的分化成子问题，而子问题与父问题要解决的问题是相同的

`flat` 方法的语法如下：

```
Array.prototype.flat([depth]);
```

`depth` 是一个可选参，如果没有提供，默认值为 `1`，如果提供了，内部会做一次 `Number` 转换，如果参数非法，`depth` 参数将会被赋值为 `0`，也就是不对数组进行拍平处理

与此同时，`flat` 方法是一个非破坏性方法，返回一个被拍平的新数组

综合以上的特性，可以得出如下的基本结构：

```
Array.prototype.flatCustom = function (depth) {
    // flat 方法返回一个新数组，所以定义一个新的数组待用
    let container = [];

    // 如果未提供参数，或者显式的提供 undefined，那么认为 depth 缺省为 1
    depth === undefined && (depth = 1);

    // 如果提供了参数，并且参数最终无法转换为 number，便认为参数非法，重新赋值为 0，即不做拍平操作
    Number.isNaN(Number(depth)) && (depth = 0);

    // 返回这个新数组
    return container;
}
```

上面已经得到了基本的结构，继续推进逻辑

对于处理递归问题，有些很关键的地方，第一个关键点就是一定要很清楚调用参数和返回值，要明白我们在解决什么问题，最终得到什么结果

第二个关键点就是问题的分化，哪些问题得到了结果不需要再分化，那些问题需要继续处理，这个需要继续处理的问题，就是递归的核心了

把思路放在数组拍平上，其核心算法就是如果一个元素不是数组，那么它就可以直接放到新数组中，这种情况就是不需要再分化并有结果的问题

下面我们来实现不再有子问题的情况

```
Array.prototype.flatCustom = function (depth) {
    let container = [];

    depth === undefined && (depth = 1);

    Number.isNaN(Number(depth)) && (depth = 0);

    // 对数组元素进行遍历，并查看元素是否是数组类型
    for (let i = 0; i < this.length; i++) {
        // 通过调用 toString 方法，查看是否是数组类型
        if (Object.prototype.toString.call(this[i]) === ['object Array']) {
```

```
        } else {
            // 如果不是数组，那么就是不存在子问题的情况，我们只需要把元素放到新数组即可
            container.push(this[i]);
        }
    }

    return container;
}
```

到此为止，算法基本上已经完成了，下一步是完成递归的核心步骤

只有在非递归步骤已经完整实现的情况下，才可以继续推进逻辑，这点非常非常重要

再来梳理一下思路，`flatCustom` 方法要解决的问题是对数组拍平，一旦元素是非数组类型，那么便可以把它放到新数组里面，这是我们已经实现的部分

再来看被暂时挂起的逻辑，如果元素是一个数组类型怎么办，重点来了，其实很简单，我们只需要对数组类型的元素继续拍平就行了，这就是递归里面所说的子问题跟父问题相同，那么如何再对子数组元素进行拍平呢，还记得 `flatCustom` 是解决什么问题的吗，没错，拍平数组，所以只需要对子数组元素再次使用该方法便可

```
Array.prototype.flatCustom = function (depth) {
    let container = [];

    depth === undefined && (depth = 1);

    Number.isNaN(Number(depth)) && (depth = 0);

    for (let i = 0; i < this.length; i++) {
        if (Object.prototype.toString.call(this[i]) === ['object Array']) {
            // flatCustom 方法在原型上
            this[i].flatCustom();
        } else {
            container.push(this[i]);
        }
    }

    return container;
}
```

现在只是处理了子问题，但是最终的目的是要把子问题的最终结果合并到父问题上的结果上，记得强调的返回值吗，没错 `flatCustom` 在设计上是要返回被拍平的数组，因此可以得出下面的逻辑：

```
Array.prototype.flatCustom = function (depth) {
    let container = [];

    depth === undefined && (depth = 1);

    Number.isNaN(Number(depth)) && (depth = 0);

    for (let i = 0; i < this.length; i++) {
        if (Object.prototype.toString.call(this[i]) === ['object Array']) {
            // flatCustom 的结果是一个被拍平的一维数组，所以需要遍历结果并把子结果合到主结果当中
            this[i].flatCustom().forEach(function(v){
                container.push(v);
```

```
            })
        } else {
            container.push(this[i]);
        }
    }

    return container;
}
```

最后再处理 `depth`，每深入一层，那么就应该在 `depth` 减一，直到 `depth` 达到边界便终止继续处理子问题，哪怕子元素还是个数组，也不再递归拍平，而是原封不动的合到父问题结果中：

```
Array.prototype.flatCustom = function (depth) {
    let container = [];

    depth === undefined && (depth = 1);

    Number.isNaN(Number(depth)) && (depth = 0);

    console.log(depth);

    for (let i = 0; i < this.length; i++) {
        if (Object.prototype.toString.call(this[i]) === '[object Array]' &&
depth > 0) {
            this[i].flatCustom(depth - 1).forEach(function (v) {
                container.push(v);
            });
        } else {
            container.push(this[i]);
        }
    }

    return container;
}
```

递归的核心便是找到最终子问题和待处理子问题，先处理好最终子问题的逻辑，再处理待处理子问题，而待处理子问题和父问题相同，那么便可以使用上面的思路进行推动。